

Типы в языках программирования

Бенджамин Пирс

28 ноября 2010 г.

Оглавление

Оглавление	3
Предисловие	11
1. Введение	21
1.1. Типы в информатике	21
1.2. Для чего годятся типы	24
1.3. Системы типов и проектирование языков	30
1.4. Краткая история	30
1.5. Дополнительная литература	31
2. Математический аппарат	35
2.1. Множества, отношения и функции	35
2.2. Упорядоченные множества	37
2.3. Последовательности	38
2.4. Индукция	39
2.5. Справочная литература	40
I Бестиповые системы	41
3. Бестиповые арифметические выражения	43
3.1. Введение	43
3.2. Синтаксис	46
3.3. Индукция на термах	49
3.4. Семантические стили	52
3.5. Вычисление	54
3.6. Дополнительные замечания	63
4. Реализация арифметических выражений на языке ML	65
4.1. Синтаксис	66
4.2. Вычисление	67
4.3. Что осталось за кадром	69

5. Бестиповое лямбда-исчисление	71
5.1. Основы	72
5.2. Программирование на языке лямбда-исчисления	77
5.3. Формальности	87
5.4. Дополнительные замечания	92
6. Представление термов без использования имен	93
6.1. Термы и контексты	94
6.2. Сдвиг и подстановка	96
6.3. Вычисление	98
7. Реализация лямбда-исчисления на ML	101
7.1. Термы и контексты	101
7.2. Сдвиг и подстановка	103
7.3. Вычисление	104
7.4. Дополнительные замечания	105
II Простые типы	107
8. Типизированные арифметические выражения	109
8.1. Типы	109
8.2. Отношение типизации	110
8.3. Безопасность = продвижение + сохранение	113
9. Простое типизированное лямбда-исчисление	119
9.1. Типы функций	119
9.2. Отношение типизации	121
9.3. Свойства типизации	124
9.4. Соотношение Карри-Говарда	128
9.5. Стирание типов и типизируемость	130
9.6. Стиль Карри и стиль Чёрча	131
9.7. Дополнительные замечания	132
10. Реализация простых типов на ML	133
10.1. Контексты	133
10.2. Термы и типы	135
10.3. Проверка типов	135
11. Простые расширения	137
11.1. Базовые типы	137
11.2. Единичный тип	138
11.3. Производные формы: последовательное исполнение и связывания-пустышки	139
11.4. Приписывание типа	141
11.5. Связывание <code>let</code>	143
11.6. Пары	145
11.7. Кортежи	147

11.8. Записи	149
11.9. Типы-суммы	152
11.10 Варианты	154
11.11 Рекурсия общего вида	160
11.12 Списки	163
12. Нормализация	169
12.1. Нормализация для простых типов	169
12.2. Дополнительные замечания	172
13. Ссылки	175
13.1. Введение	176
13.2. Типизация	180
13.3. Вычисление	181
13.4. Типизация содержимого памяти	184
13.5. Безопасность	187
13.6. Дополнительные замечания	190
14. Исключения	193
14.1. Порождение исключений	194
14.2. Обработка исключений	195
14.3. Исключения, сопровождаемые значениями	197
III Подтипы	203
15. Подтипы	205
15.1. Включение	205
15.2. Отношение подтиповирования	206
15.3. Свойства подтипов и типизации	212
15.4. Типы <i>Top</i> и <i>Bottom</i>	216
15.5. Подтипы и другие элементы языка	218
15.6. Семантика подтипов, основанная на преобразованиях типов	225
15.7. Типы-пересечения и типы-объединения	231
15.8. Дополнительные замечания	232
16. Метатеория подтипов	235
16.1. Алгоритмическое отношение подтиповирования	236
16.2. Алгоритмическое отношение типизации	240
16.3. Пересечения и объединения	245
16.4. Алгоритмическая типизация и тип <i>Bot</i>	247
17. Реализация подтипов на ML	249
17.1. Синтаксис	249
17.2. Подтипы	249
17.3. Типизация	250

18. Расширенный пример: императивные объекты	253
18.1. Что такое объектно-ориентированное программирование?	253
18.2. Объекты	256
18.3. Генераторы объектов	257
18.4. Подтипы	257
18.5. Группировка переменных экземпляра	258
18.6. Простые классы	258
18.7. Добавление новых переменных экземпляра	261
18.8. Вызов методов надкласса	262
18.9. Классы с переменной <code>self</code>	262
18.10 Открытая рекурсия через <code>self</code>	263
18.11 Открытая рекурсия и порядок вычислений	264
18.12 Более эффективная реализация	269
18.13 Резюме	271
18.14 Дополнительные замечания	272
19. Расширенный пример: Облегченная Java	275
19.1. Введение	275
19.2. Обзор	277
19.3. Именные и структурные системы типов	279
19.4. Определения	282
19.5. Свойства	289
19.6. Объекты: кодирование или элементарное понятие?	291
19.7. Дополнительные замечания	291
IV Рекурсивные типы	295
20. Рекурсивные типы	297
20.1. Примеры	298
20.2. Формальные определения	305
20.3. Подтипы	308
20.4. Дополнительные замечания	309
21. Метатеория рекурсивных типов	311
21.1. Индукция и коиндукция	312
21.2. Конечные и бесконечные типы	314
21.3. Подтипы	316
21.4. Отступление о транзитивности	319
21.5. Проверка принадлежности	320
21.6. Более эффективные алгоритмы	325
21.7. Регулярные деревья	328
21.8. μ -типы	330
21.9. Подсчет подвыражений	334
21.10 Отступление: экспоненциальный алгоритм	339
21.11 Подтиповирование для изорекурсивных типов	341
21.12 Дополнительные замечания	342

V Полиморфизм	345
22. Реконструкция типов	347
22.1. Типовые переменные и подстановки	347
22.2. Две точки зрения на типовые переменные	349
22.3. Типизация на основе ограничений	350
22.4. Унификация	356
22.5. Главные типы	359
22.6. Неявные аннотации типов	360
22.7. Полиморфизм через <code>let</code>	360
22.8. Дополнительные замечания	366
23. Универсальные типы	369
23.1. Мотивация	369
23.2. Разновидности полиморфизма	370
23.3. Система F	371
23.4. Примеры	373
23.5. Основные свойства	382
23.6. Стирание, типизируемость и реконструкция типов	382
23.7. Стирание и порядок вычислений	386
23.8. Варианты Системы F	387
23.9. Параметричность	388
23.10. Импредикативность	389
23.11. Дополнительные замечания	390
24. Экзистенциальные типы	393
24.1. Мотивация	393
24.2. Абстракция данных при помощи экзистенциальных типов	397
24.3. Кодирование экзистенциальных типов	407
24.4. Дополнительные замечания	409
25. Реализация Системы F на ML	411
25.1. Представление типов без использования имен	411
25.2. Сдвиг типов и подстановка	412
25.3. Термы	413
25.4. Вычисление	415
25.5. Типизация	416
26. Ограниченнная квантификация	419
26.1. Мотивация	419
26.2. Определения	421
26.3. Примеры	426
26.4. Безопасность	430
26.5. Ограниченные экзистенциальные типы	436
26.6. Дополнительные замечания	438
27. Расширенный пример: еще раз императивные объекты	441

28.Метатеория ограниченной квантификации	447
28.1. Выявление	447
28.2. Минимальная типизация	448
28.3. Подтипы в ядерной $F_{<:}$	451
28.4. Подтипы в полной $F_{<:}$	454
28.5. Неразрешимость полной $F_{<:}$	457
28.6. Объединения и пересечения	462
28.7. Ограниченные кванторы существования	465
28.8. Ограниченнaя квантификация и тип <code>Bot</code>	466
VI Системы высших порядков	467
29.Операторы над типами и виды	469
29.1. Неформальное введение	470
29.2. Определения	476
30.Полиморфизм высших порядков	479
30.1. Определения	479
30.2. Пример	480
30.3. Свойства исчисления	481
30.4. Варианты F_ω	489
30.5. Идем дальше: зависимые типы	490
31.Подтипы высших порядков	499
31.1. Интуитивные понятия	500
31.2. Определения	501
31.3. Свойства исчисления	502
31.4. Дополнительные замечания	502
32.Расширенный пример: чисто функциональные объекты	507
32.1. Простые объекты	507
32.2. Подтипы	508
32.3. Ограниченнaя квантификация	509
32.4. Типы интерфейсов	511
32.5. Отправка сообщений объектам	512
32.6. Простые классы	513
32.7. Полиморфные обновления	514
32.8. Добавление переменных экземпляра	517
32.9. Классы с переменной <code>self</code>	518
32.10Дополнительные замечания	519
Приложения	525
А. Решения избранных упражнений	525

В. Принятые обозначения	595
В. Имена метапеременных	595
В. Имена правил	596
В. Соглашения по именам и индексам	596
Литература	597
Список иллюстраций	637
Предметный указатель	640

Предисловие

Исследование систем типов, а также языков программирования с точки зрения теории типов, в последнее время превратилось в энергично развивающуюся научную дисциплину, которая имеет важные приложения в проектировании программ, в разработке языков программирования, в реализации высокопроизводительных компиляторов и в области безопасности. Настоящая книга представляет собой подробное введение в основные понятия, достижения и методы этой дисциплины.

Предполагаемая аудитория

Книга предназначена для двух основных групп читателей: с одной стороны, для аспирантов и исследователей, которые специализируются на языках программирования и теории типов; с другой стороны, для аспирантов и продвинутых студентов из других областей информатики, которые хотели бы изучить основные понятия теории языков программирования. Первой группе книга дает обширный обзор рассматриваемой области, достаточно глубокий, чтобы позволить чтение исследовательской литературы. Второй группе она предоставляет подробный вводный материал и обширное собрание примеров, упражнений и практических ситуаций. Книга может использоваться как в вводных курсах аспирантского уровня, так и в более продвинутых семинарах по языкам программирования.

Цели

Основной моей задачей был **охват** основных тем, включая базовую операционную семантику и связанные с ней способы построения доказательств, бестиповое лямбда-исчисление, простые системы типов, универсальный и экзистенциальный полиморфизм, реконструкцию типов, подтипы, ограниченную квантификацию, рекурсивные типы и операторы над типами, а также краткое обсуждение некоторых других вопросов.

Второй задачей был **pragmatism**. Книга сосредоточена на использовании систем типов в языках программирования и пренебрегает некоторыми темами (например, денотационной семантикой), которые, вероятно, были бы включены в более математически-ориентированный текст по типизированным лямбда-исчислениям. В качестве базового вычислительного формализма выбрано лямбда-исчисление с вызовом по значению, которое используется в большинстве современных языков программирования и легко расширяется

императивными конструкциями вроде ссылок и исключений. При рассмотрении каждой возможности языка обсуждаются такие вопросы, как: практическая *мотивация* для введения этой возможности; методы доказательства *безопасности* языков, содержащих эту возможность; а также трудности, с которыми приходится сталкиваться при её *реализации* — в особенности, построение и анализ алгоритмов проверки типов.

В число моих задач также входило уважение к **разнообразию** в рассматриваемой дисциплине; книга охватывает множество отдельных тем и несколько хорошо исследованных их сочетаний, но при этом не предпринимается попыток собрать все в единую целостную систему. Существуют единые подходы к некоторым подмножествам набора тем — например, многие разновидности «функциональных типов» можно компактно и изящно описать в единообразной нотации *чистых систем типов*. Однако, дисциплина в целом пока что развивается столь быстро, что систематизировать ее целиком невозможно.

Книга построена так, чтобы **облегчить ее использование**, как в рамках университетских курсов, так и при самостоятельном изучении. К большинству упражнений прилагаются полные решения. Базовые определения для удобства организованы в виде таблиц. Зависимости между понятиями и системами по возможности указываются явно. К тексту прилагается обширная библиография и предметный указатель.

Наконец, последним организующим принципом при написании книги была **честность**. Все обсуждаемые в книге системы (кроме некоторых, упомянутых лишь мельком) реализованы на практике. Для каждой главы существует программа проверки типов и интерпретатор, и все примеры были механически проверены с их помощью.* Эти программы доступны на веб-сайте книги, их можно использовать при программировании упражнений, для экспериментов по дальнейшему расширению и в учебных проектах.

Чтобы достичь поставленных целей, пришлось пожертвовать некоторыми другими аспектами книги. Наиболее важное из них — **полнота** рассмотрения. Полный обзор дисциплины языков программирования и систем типов, вероятно, невозможен в рамках одного текста — тем более, в рамках учебника. Я сосредоточился на тщательном рассмотрении базовых понятий; многочисленные ссылки к научной литературе служат отправными точками для дальнейших исследований. Также не была целью практическая **эффективность** алгоритмов проверки типов: эта книга не является пособием по реализации компиляторов или средств проверки типов промышленного уровня.

Структура книги

В части I обсуждаются бестиповые системы. Основные понятия абстрактного синтаксиса, индуктивные определения и доказательства, правила вывода и операционная семантика сначала вводятся в контексте чрезвычайно простого языка для работы с числами и логическими значениями, а затем повторяются для бестипового лямбда-исчисления. В части II рассматривается простое типизированное лямбда-исчисление, а также набор базовых языковых конструкций: типы-произведения, типы-суммы, записи, варианты, ссылки и

*При верстке перевода это правило не соблюдалось. — *прим. перев.*

исключения. Вводная глава о типизированных арифметических выражениях плавно подводит к ключевой идее типовой безопасности. В одной из глав (которую можно пропустить) методом Тейта доказывается теорема о нормализации для простого типизированного лямбда-исчисления. Часть III посвящена основополагающему механизму подтипов; она содержит подробное обсуждение метатеории и два расширенных примера. В части IV рассматриваются курсивные типы, как в простой *изорекурсивной* формулировке, так и в более хитроумной *эквирекурсивной*. Вторая глава этой части развивает метатеорию системы с эквирекурсивными типами и подтипов в рамках математического метода коиндукции. Темой части V является полиморфизм. Она содержит главы о реконструкции типов в стиле ML, о более мощном импредикативном полиморфизме Системы F, об экзистенциальной квантификации и ее связям с абстрактными типами данных, а также о комбинации полиморфизма и подтипов в системах с ограниченной квантификацией. В части VI речь идет об операторах над типами. В первой главе вводятся основные понятия; в следующей разрабатывается Система F_ω и ее метатеория; в третьей операторы над типами скрещиваются с ограниченной квантификацией, давая в результате Систему F_{ζ}^ω ; последняя глава представляет собой расширенный пример.

Основные зависимости между главами изображены на рис. 1. Серые стрелки означают, что только часть более поздней главы зависит от более ранней.

Обсуждение каждой языковой конструкции, представленной в книге, следует общей схеме. Сначала идут мотивирующие примеры; затем излагаются формальные определения; затем приводятся доказательства основных свойств, таких как типовая безопасность; затем (обычно в отдельной главе) следует более глубокое исследование метатеории, которое ведет к алгоритмам проверки типов и доказательству их корректности, полноты и гарантии завершения; наконец (опять же, в отдельной главе) приводится конкретная реализация этих алгоритмов в виде программы на языке OCaml (Objective Caml).

На протяжении всей книги важным источником примеров служит анализ и проектирование языковых возможностей для объектно-ориентированного программирования. В четырех главах с расширенными примерами детально развиваются различные подходы: простая модель с обычными императивными объектами и классами (глава 18), базовое исчисление, основанное на Java (глава 19), более тонкий подход к императивным объектам с использованием ограниченной квантификации (глава 27), а также рассмотрение объектов и классов в рамках чисто функциональной Системы F_{ζ}^ω , при помощи экзистенциальных типов (глава 32).

Чтобы материал книги можно было охватить в рамках односеместрового продвинутого курса — а саму книгу мог поднять средний аспирант, — пришлось исключить из рассмотрения многие интересные и важные темы. Денотационный и аксиоматический подходы к семантике опущены полностью; по этим направлениям уже существуют замечательные книги, и эти темы отвлекали бы от строго pragматической, ориентированной на реализацию, перспективы, принятой в этой книге. В нескольких местах упоминаются богатые связи между системами типов и логикой, но в детали я не вдаюсь; эти связи важны, но они ушли бы нас слишком далеко в сторону. Многие продвинутые возмож-

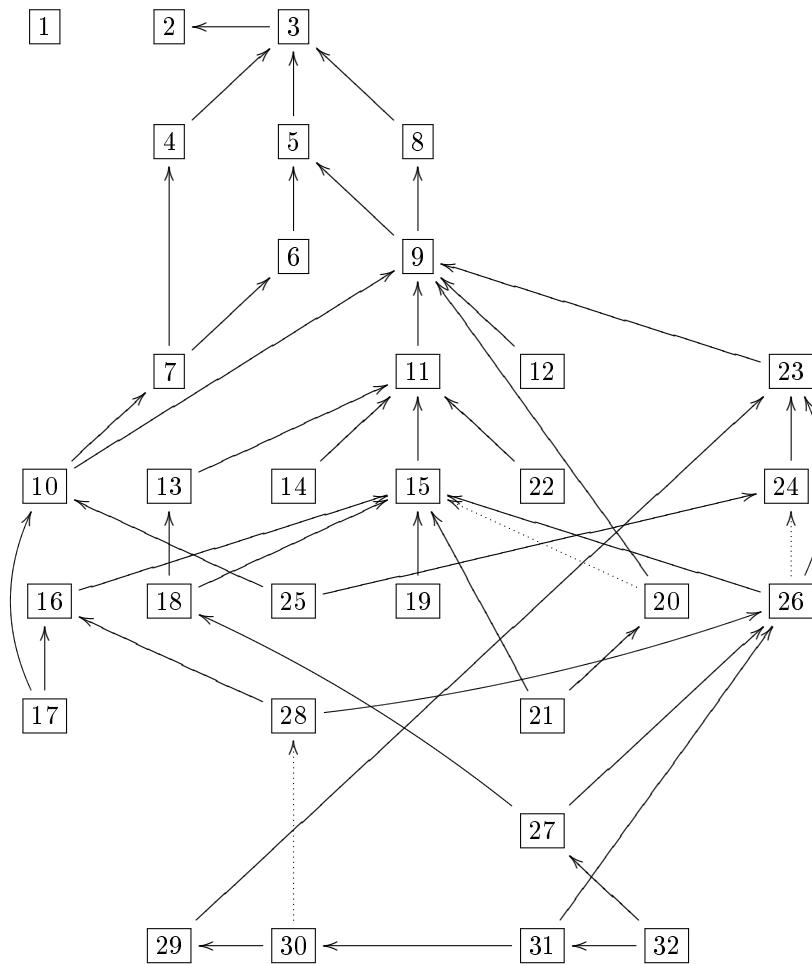


Рис. 1. Зависимости между главами

ности языков программирования и систем типов упомянуты лишь мельком (например, зависимые типы, типы-пересечения, а также соотношение Карри-Говарда); краткие разделы по этим вопросам служат отправными пунктами для дальнейшего самостоятельного изучения. Наконец, не считая краткого экскурса в Java-подобный базовый язык (глава 19), в книге рассматриваются исключительно языки на основе лямбда-исчисления; однако понятия и механизмы, исследованные в этих рамках, могут быть напрямую перенесены в соседние области, такие как типизированные параллельные языки, типизированные ассемблеры и специализированные исчисления объектов.

Требуемая подготовка

Данный текст не предполагает никаких знаний по теории языков программирования, однако читатель должен в известной степени обладать математической зрелостью — а именно, иметь опыт строгой учебной работы в таких областях, как дискретная математика, алгоритмы и элементарная логика.

Читатель должен быть знаком, по меньшей мере, с одним функциональным языком высокого уровня (Scheme, ML, Haskell и т. п.) и с основными понятиями теории языков программирования и компиляторов (абстрактный синтаксис, BNF-грамматики, вычисление, абстрактные машины и т. п.). Этот материал доступен для изучения во множестве замечательных учебников; я особенно ценю «Основные понятия языков программирования» Фридмана, Ванда и Хейнса («*Essentials of Programming Languages*», Friedman, Wand, and Haynes, 2001), а также «Прагматику языков программирования» Скотта («*Programming Language Pragmatics*», Scott, 1999). В нескольких главах пригодится опыт работы с каким-либо объектно-ориентированным языком, например, с Java (Arnold and Gosling, 1996).

Главы, посвященные конкретным реализациям программ проверки типов, содержат большие фрагменты кода на OCaml (он же Objective Caml), популярном диалекте языка ML. Для чтения этих глав полезно, но не абсолютно необходимо, знать OCaml; в тексте используется только небольшое подмножество языка, и его конструкции объясняются при их первом использовании. Эти главы образуют отдельную нить в ткани книги, и при желании их можно полностью пропустить.

В настоящее время лучший учебник по OCaml — книга Кузино и Мони (Cousineau and Mauny, 1998). Кроме того, вполне пригодны для чтения и учебные материалы, которые поставляются с дистрибутивом OCaml (<http://caml.inria.fr> или <http://www.ocaml.org>).

Для читателей, знакомых с другим крупным диалектом ML, Standard ML, фрагменты кода на OCaml не должны представлять трудности. Среди популярных учебников по Standard ML можно назвать книги Поульсона (Paulson, 1996) и Ульмана (Ullman, 1997).

Примерный план курса

В рамках аспирантского курса среднего или продвинутого уровня можно пройти большую часть материала книги за семестр. На рис. 2 показана примерная схема продвинутого курса для аспирантов Пенсильванского университета (две лекции по 90 минут в неделю; предполагается наличие минимальной подготовки в теории языков программирования, однако продвижение очень быстрое).

При составлении курса для студентов или вводного курса для аспирантов можно выбрать несколько способов подачи материала. Курс по *системам типов в программировании* будет сосредоточен на главах, в которых вводятся различные особенности типовых систем и иллюстрируется их использование; большая часть метатеории и главы, в которых строятся реализации, можно пропустить. С другой стороны, в курсе по *основам теории и реализации систем типов* можно использовать все ранние главы, возможно, за исключе-

ЛЕКЦИЯ	ТЕМА	МАТЕРИАЛ
1.	Обзор курса; история; административные формально-сти	1, (2)
2.	Предварительная информация: синтаксис, операционная семантика	3, 4
3.	Введение в лямбда-исчисление	5.1, 5.2
4.	Формализация лямбда-исчисления	5.3, 6.7
5.	Типы; простое типизированное лямбда-исчисление	8, 9, 10
6.	Простые расширения; производные формы	11
7.	Другие расширения	11
8.	Нормализация	12
9.	Ссылки; исключения	13, 14
10.	Подтипы	15
11.	Метатеория подтипов	16, 17
12.	Императивные объекты	18
13.	Облегченная Java	19
14.	Рекурсивные типы	20
15.	Метатеория рекурсивных типов	21
16.	Метатеория рекурсивных типов	21
17.	Реконструкция типов	22
18.	Универсальный полиморфизм	23
19.	Экзистенциальный полиморфизм; АТД	24, (25)
20.	Ограниченнная квантификация	26, 27
21.	Метатеория ограниченной квантификации	28
22.	Операторы над типами	29
23.	Метатеория F_ω	30
24.	Подтипы высших порядков	31
25.	Чисто функциональные объекты	32
26.	Лекция про запас	

Рис. 2. Примерная схема продвинутого аспирантского курса

нием главы 12 (еще, может быть, 18 и 21), принося в жертву более сложный материал в конце книги. Можно также построить более короткие курсы, выбирая отдельные главы и следуя диаграмме на рис. 1.

Книгу также можно использовать как базовый текст более широко организованного аспирантского курса по теории языков программирования. В таком курсе можно потратить половину или две трети семестра на изучение большей части книги, а остаток времени посвятить, скажем, рассмотрению теории параллелизма на основе книги Милнера по пи-исчислению (Milner, 1999), введению в логику Хоара и аксиоматическую семантику (напр., Winskel, 1993) или обзору продвинутых языковых конструкций вроде продолжений или систем модулей.

Если существенная роль отводится курсовым проектам, может оказаться полезным отложить часть теоретического материала (скажем, нормализацию, и, возможно, некоторые главы по метатеории), чтобы продемонстрировать широкий спектр примеров, прежде чем студенты выберут себе темы курсовых.

Упражнения

Большинство глав снабжено многочисленными примерами. Некоторые рассчитаны на решение с помощью карандаша и бумаги, некоторые требуют программирования в обсуждаемых исчислениях, а некоторые касаются реализации этих исчислений на языке ML. Ориентировочная сложность упражнений указывается по следующей шкале:

★	Контрольный вопрос	от 30 секунд до 5 минут
★★	Простое	≤ 1 час
★★★	Средней сложности	≤ 3 часа
★★★★	Тяжелое	> 3 часов

Упражнения, отмеченные знаком ★, предназначены для быстрой проверки понимания важных концепций. Я советую читателю останавливаться и решать их, прежде чем продолжать чтение. В каждой главе словом РЕКОМЕНДУЕТСЯ отмечен набор упражнений размером примерно с домашнее задание.

В приложении A приведены полные решения большинства упражнений. Чтобы избавить читателя от напрасного поиска в тех немногих случаях, когда их нет, используется значок ↗.

Типографские соглашения

Большинство глав книги построено так: сначала особенности некоторой системы типов описываются в тексте, а затем эта система определяется формально в виде набора правил вывода, сведенных в одну или несколько таблиц. Чтобы облегчить их использование в качестве справочника, эти определения обычно приводятся полностью, включая не только новые правила для возможностей, обсуждаемых в данный момент, но и остальные, сколько их требуется для построения полного исчисления. Новые фрагменты системы печатаются на сером фоне, чтобы отличия от предыдущих систем были более заметны.

Необычной особенностью этой книги является то, что все примеры были механически проверены при верстке: с помощью особого скрипта из каждой главы извлекались примеры, строилась и компилировалась специальная программа проверки типов, содержащая обсуждаемые в главе особенности, эта программа прогонялась на примерах, и результаты проверки вставлялись в текст.* Всю сложную работу выполняла при этом система под названием *TinkerType*, написанная нами с Майклом Левином (Levin and Pierce, 2001). Средства на ее разработку были получены от Национального Научного Фонда по грантам CCR-9701826 «Принципиальные основы программирования с объектами» и CCR-9912352 «Модульные системы типов».

Электронные ресурсы

Веб-сайт, посвященный этой книге, можно найти по адресу

<http://www.cis.upenn.edu/~bcpierce/tapl>

* При верстке перевода это правило не соблюдалось. — прим. перев.

На этом сайте расположены следующие ресурсы: список ошибок, найденных в тексте, предложения по возможным курсовым проектам, ссылки на дополнительный материал, предоставленный читателями, а также набор реализаций (программы проверки типов и простые интерпретаторы) для исчислений, рассмотренных в каждой из глав книги.

Эти реализации представляют собой среду для экспериментов с примерами, приведенными в книге, и для проверки ответов на упражнения. Они написаны с упором на удобство чтения и возможность модификаций. Слушатели моих курсов успешно использовали их в качестве основы как для простых упражнений, так и для курсовых проектов более солидного размера. Эти реализации написаны на OCaml. Бесплатный компилятор OCaml можно найти по адресу <http://caml.inria.fr>; на большинстве платформ он устанавливается безо всякой труда.

Читателям стоит также знать о существовании списка рассылки Types Forum, посвященного всем аспектам систем типов и их реализаций. Список модерируется, что позволяет поддерживать относительную компактность и высокое отношение полезного сигнала к шуму. Архивы рассылки и инструкции для подписчиков можно найти по адресу <http://www.cis.upenn.edu/~bcpierce/types>.

Благодарности

Читатели, которые найдут эту книгу полезной, должны прежде всего быть благодарны четырем учителям — Луке Карделли, Бобу Харперу, Робину Милнеру и Джону Рейнольдсу, — которые научили меня почти всему, что я знаю о языках программирования и типах.

Остальные знания я приобрел в основном в совместных проектах; помимо Луки, Боба, Робина и Джона, среди моих партнеров в этих проектах были Мартин Абади, Гордон Плоткин, Рэнди Поллак, Дэвид Н. Тёрнер, Диана Реми, Давиде Санджорджи, Адриана Компаньони, Мартин Хоффман, Джузеппе Кастанья, Мартин Стеффен, Ким Брюс, Наоки Кобаяси, Харуо Хосоя, Ацуси Игараси, Филип Уодлер, Питер Бьюнеман, Владимир Гапеев, Майкл Левин, Питер Сьюэлл, Джером Вуйон и Эйдзиро Сумии. Сотрудничество с ними послужило для меня не только источником понимания, но и удовольствия от работы над этой темой.

Структура и организация этого текста стали лучше в результате педагогических консультаций с Торстеном Альтеркирхом, Бобом Харпером и Джоном Рейнольдсом, а сам текст выиграл от замечаний и исправлений, авторами которых были Джим Александер, Джош Бердин, Тони Боннер, Джон Танг Бойланд, Дэйв Кларк, Диего Дайнезе, Оливье Данви, Мэттью Дэвис, Владимир Гапеев, Боб Харпер, Эрик Хилсдейл, Харуо Хосоя, Ацуси Игараси, Роберт Ирвин, Такаясу Ито, Асаф Кфури, Майкл Левин, Василий Литвинов, Пабло Лопес Оливас, Дэйв Маккуин, Нарсисо Марти-Олиет, Филипп Менье, Робин Милнер, Матти Нуокянен, Гордон Плоткин, Джон Превост, Фермин Рейг, Диана Реми, Джон Рейнольдс, Джеймс Рили, Охад Роде, Юрген Шлегельмильх, Алан Шмитт, Эндрю Схонмакер, Олин Шиверс, Педрита Стивенс, Крис Стоун, Эйдзиро Сумии, Вэл Тэннен, Джером Вуйон и Филип Уодлер (я прошу

прощения, если кого-то случайно забыл включить в этот список). Лука Карделли, Роджер Хиндли, Дэйв Маккуин, Джон Рейнольдс и Джонатан Селдин поделились исторической перспективой некоторых запутанных вопросов.

Участники моих аспирантских семинаров в Индианском университете в 1997 и 1998 годах и в Пенсильванском университете в 1999 и 2000 годах работали с ранними версиями этой книги; их мнения и комментарии помогли мне придать ей окончательную форму. Боб Прайор и его сотрудники в издательстве «The MIT Press» весьма профессионально провели рукопись через все многочисленные стадии публикации. Дизайн книги основан на макросах \LaTeX , которые разработал для «The MIT Press» Кристофер Манинг.

Доказательства программ
настолько скучны, что
социальные механизмы
математики на них не работают.

Ричард Де Милло, Ричард
Липтон и Аллан Перлис, 1979

... Поэтому при верификации
не стоит рассчитывать на
социальные механизмы.

Дэвид Дилл, 1999

Формальные методы не будут
принести существенных
результатов до тех пор, пока их
не смогут использовать люди,
которые их не понимают.

приписывается Тому Мелхэму

Глава 1

Введение

1.1. Типы в информатике

Современные технологии программного обеспечения располагают широким репертуаром *формальных методов* (formal methods), которые помогают убедиться, что система ведет себя в соответствии с некоторой спецификацией ее поведения, явной или неявной. На одном конце шкалы находятся мощные конструкции, такие как логика Хоара, языки алгебраической спецификации, модальные логики и денотационные семантики. Все они способны выразить самые широкие требования к корректности программ, однако часто очень трудны в использовании и требуют от программистов высочайшей квалификации. На другом конце спектра располагаются намного более скромные методы — настолько скромные, что автоматические алгоритмы проверки могут быть встроены в компиляторы, компоновщики или автоматические анализаторы программ, а применять их могут даже программисты, не знакомые с теоретическими основами этих методов. Хорошо известный пример таких облегченных *формальных методов* (lightweight formal method) — *программы проверки моделей* (model checkers) — инструменты для поиска ошибок в таких конечных системах, как интегральные схемы или коммуникационные протоколы. Другой пример, приобретающий сейчас популярность — *мониторинг во время исполнения* (run-time monitoring), набор приемов, позволяющих системе динамически обнаруживать, что поведение одного из ее компонентов отклоняется от спецификации. Однако же, самый популярный и испытанный облегченный формальный метод — это *системы типов* (type systems), которым в основном и посвящена эта книга.

Как и многие другие термины, используемые в больших сообществах, термин «система типов» трудно определить так, чтобы охватить его неформальное использование в среде разработчиков и авторов языков программирования, но чтобы при этом определение было достаточно конкретным. Вот, например, одно из возможных определений:

Система типов — это гибко управляемый синтаксический метод доказательства отсутствия в программе определенных видов

поведения при помощи классификации выражений языка по разновидностям вычисляемых ими значений.

Некоторые моменты заслуживают дополнительного пояснения. Во-первых, в этом определении системы типов названы средством для рассуждений о *программах*. Такой выбор слов отражает ориентацию этой книги на системы типов, используемые в языках программирования. В более общем смысле, термин *системы типов* (или *теория типов*, type theory) относится к намного более обширной области исследований в логике, математике и философии. В этом смысле типы были впервые формально описаны в начале 1900-х годов как средство избежать логических парадоксов, угрожавших основаниям математики, например, парадокса Рассела ([Russell, 1902](#)). В течение двадцатого века типы превратились в стандартный инструмент логики, особенно в теории доказательств (см. [Gandy, 1976](#) и [Hindley, 1997](#)), и глубоко проникли в язык философии и науки. В этой области основными достижениями были *теория типов Рассела* (ramified theory of types) ([Whitehead and Russell, 1910](#)), *простая теория типов* (simple theory of types) Рамсея ([Ramsey, 1925](#)) — основа для простого типизированного лямбда-исчисления Чёрча ([Church, 1940](#)), *конструктивная теория типов* (constructive theory of types) Мартина-Лёфа ([1973, 1984](#)) и *чистые системы типов* (pure type systems) Бенарди, Терлоу и Барендргта ([Berardi, 1988; Terlouw, 1989; Barendregt, 1992](#)).

Внутри самой информатики как научной дисциплины изучение систем типов разделяется на две ветви. Основной темой этой книги является более практическая ветвь, в которой исследуются приложения к языкам программирования. Более абстрактная ветвь изучает соответствия между различными «чистыми типизированными лямбда-исчислениями» и разновидностями логики, через *изоморфизм Карри-Говарда* (Curry-Howard correspondence) ([§9.4](#)). В обоих сообществах используются аналогичные понятия, системы записи и методы, однако есть важные различия в подходе. Например, исследования типизированного лямбда-исчисления обычно имеют дело с системами, в которых для всякого правильно типизированного вычисления гарантировано завершение, в то время как большинство языков программирования жертвуют этим свойством ради таких инструментов, как рекурсивные функции.

Еще одно важное свойство вышеуказанного определения — упор на *классификацию* термов (синтаксических составляющих) в соответствии со значениями, которые они порождают, будучи вычисленными. Систему типов можно рассматривать как *статическую* (static) аппроксимацию поведения программы во время выполнения. (Более того, типы термов обычно вычисляются *композиционально* (compositionally), то есть тип выражения зависит только от типов его подвыражений.)

Иногда слово «статический» добавляется явным образом — например, говорят о «языках программирования со статической типизацией», — чтобы отличить тот анализ, который производится при компиляции, и который мы рассматриваем здесь, от *динамической* (dynamic) или *латентной* (latent) типизации в языках вроде Scheme ([Sussman and Steele, 1975; Kelsey, Clinger, and Rees, 1998; Dybvig, 1996](#)), в которых *теги типов* (type tags) используются для различия видов объектов, находящихся в куче. Термин «динамически типизированный», по нашему мнению, неверен (его следовало бы заменить на

«динамически проверяемый»), но такое употребление уже общепринято.

Будучи статическими, системы типов обязательно *консервативны* (*conservative*): они способны однозначно доказать *отсутствие* определенных нежелательных видов поведения, но не могут доказать их наличие, и, следовательно, иногда вынуждены отвергать программы, которые на самом деле при выполнении ведут себя корректным образом. Например, программа

```
if <сложная проверка> then S else <ошибка типа>
```

будет отвергнута как неверно типизированная, даже если <сложная проверка> всегда выдает значение «истина», поскольку статический анализ неспособен это заметить. Баланс между консервативностью и выразительностью — фундаментальный фактор при разработке систем типов. Стремление типизировать как можно большее число программ путем присвоения их выражениям все более точно определенных типов — основная движущая сила в данной научной дисциплине.

Аналогично, относительно незамысловатые виды анализа, воплощенные в большинстве систем типов, неспособны запретить совсем любое нежелательное поведение программ; они гарантируют только отсутствие в программах *определенных* видов ошибок. Например, большинство систем типов могут статически проверить, что в качестве аргументов элементарных арифметических операций всегда используются числа, что объект-получатель в вызове метода всегда имеет соответствующий метод, и т. п., но не могут обеспечить, чтобы делитель всегда не был равен нулю или чтобы индексы массива никогда не выходили за предельные значения.

Виды некорректного поведения, которые в каждом конкретном языке могут быть устранины при помощи типов, часто называют *ошибками типов времени выполнения* (*run-time type errors*). Важно помнить, что набор таких ошибок в каждом языке свой: несмотря на то, что наборы ошибок времени выполнения в разных языках существенно пересекаются, в принципе каждая система типов сопровождается описанием ошибок, которые она должна предотвращать. *Безопасность* (*safety*) (или *корректность*, *soundness*) системы типов должна определяться по отношению к её собственному набору ошибок времени выполнения.

Виды неправильного поведения, выявляемые через анализ типов, не ограничиваются низкоуровневыми ошибками вроде вызова несуществующих методов: системы типов используют также для обеспечения высокоуровневой *модульности* (*modularity*) и для защиты целостности *абстракций* (*abstractions*), определяемых пользователем. Нарушение скрытия информации (например, прямое обращение к полям объекта, реализация которого должна быть абстрактной) является ошибками типизации в той же мере, как и, скажем, использование целого числа как указателя, что приводит к аварийному завершению работы программы.

Процедуры проверки типов обычно встроены в компиляторы или компоновщики. Следовательно, они должны уметь выполнять свою работу *автоматически* (*automatically*), без ручного вмешательства или взаимодействия с программистом, — т. е., они должны содержать вычислительно *разрешимые* (*tractable*) алгоритмы анализа. Однако, это по-прежнему оставляет программисту множество способов повлиять на работу анализатора с помощью явных

аннотаций типов (type annotations) в программах. Обычно эти аннотации делают достаточно простыми, чтобы программы было легче писать и читать. В принципе, однако, в аннотациях типов можно закодировать полное доказательство соответствия программы некоторой произвольной спецификации; в этом случае, программа проверки типов превращается в программу проверки *доказательств* (proof checker). Такие технологии, как Extended Static Checking («расширенная статическая проверка») (Detlefs, Leino, Nelson, and Saxe, 1998), наводят мосты между системами типов и методами всеобъемлющей верификации программ. Эти технологии реализуют полностью автоматизированную проверку некоторого широкого класса желательных свойств, используя для работы лишь «достаточно легковесные» аннотации.

Ещё заметим, что нам прежде всего интересны методы, которые не просто в принципе поддаются автоматизации, но которые обеспечивают *эффективные* алгоритмы проверки типов. Впрочем, вопрос о том, что именно считать эффективным, остается открытым. Даже в широко используемых системах типов, вроде ML (Damas and Milner, 1982), время проверки в некоторых патологических случаях может быть громадным (Henglein and Mairson, 1991). Имеются даже языки, в которых задача проверки или реконструкции типов не является разрешимой, но в которых есть алгоритмы, ведущие к быстрому останову «в большинстве случаев, представляющих практический интерес» (напр., Pierce and Turner, 2000; Nadathur and Miller, 1988; Pfenning, 1994).

1.2. Для чего годятся типы

Выявление ошибок

Самое очевидное достоинство статической проверки типов — это то, что она помогает раньше обнаруживать некоторые ошибки в программах. Рано обнаруженные ошибки могут быть немедленно исправлены, а не прятаться долго в коде, чтобы потом неожиданно всплыть, когда программист занят чем-то совершенно другим — или даже после того, как программа передана пользователям. Более того, зачастую место возникновения ошибки можно точнее определить при проверке типов, а не во время выполнения, когда их последствия могут обнаружиться не сразу, а лишь спустя некоторое время после того, как программа начинает работать неправильно.

На практике статическая проверка типов обнаруживает удивительно широкий спектр ошибок. Программисты, работающие в языках с богатой системой типов, замечают, что их программы зачастую «сразу начинают работать», если проходят проверку типов, причем происходит это даже чаще, чем они сами могли бы бы ожидать. Одно из возможных объяснений состоит в том, что в виде несоответствий на уровне типов часто проявляются не только тривиальные неточности (скажем, когда программист забывает преобразовать строку в число, прежде чем извлечь квадратный корень), но и более глубокие концептуальные ошибки (например, пропуск граничного условия в сложном разборе случаев, или смешение единиц измерения в научном вычислении). Сила этого эффекта зависит от выражительности системы типов и от решаемой программистской задачи: программы, работающие с большим набором структур дан-

ных (скажем, компиляторы) дают больше возможностей для проверки типов, чем программы, которые работают лишь с несколькими простыми типами, скажем, научные вычислительные задачи (хотя и тут могут оказаться полезны тонкие системы типов, поддерживающие *анализ размерностей* (dimension analysis); см. Kennedy, 1994).

Извлечение максимальной выгоды из системы типов, как правило, требует от программиста некоторого внимания и готовности в полной мере использовать возможности языка; например, если в сложной программе все структуры данных представлены в виде списков, то компилятор не сможет помочь в полной мере, как если бы для каждой из них определялся свой тип данных или абстрактный тип. Выразительные системы типов предоставляют специальные «трюки», позволяющие выражать информацию о структуре данных в виде типов.

Для некоторых видов программ процедура проверки типов может служить инструментом *сопровождения* (maintenance). Например, программист, которому требуется изменить определение сложной структуры, может не искать вручную все места в программе, где требуется изменить код, имеющий дело с этой структурой. Как только изменяется определение типа данных, во всех этих фрагментах кода возникают ошибки типов, и их можно найти путем простого прогона компилятора и исправления тех мест, где проверка типов не проходит.

Абстракция

Еще одно важное достоинство использования систем типов при разработке программ — это поддержание дисциплины программирования. В частности, в контексте построения крупномасштабных программных систем, системы типов являются стержнем *языков описания модулей* (module languages), при помощи которых компоненты больших систем упаковываются и связываются воедино. Типы появляются в интерфейсах модулей (или близких по смыслу структур, таких, как классы); в сущности, сам интерфейс можно рассматривать как «тип модуля», содержащий информацию о возможностях, которые модуль предоставляет — как своего рода частичное соглашение между разработчиками и пользователями.

Разбиение больших систем на модули с ясно определенными интерфейсами приводит к более абстрактному стилю проектирования, в котором интерфейсы разрабатываются и обсуждаются отдельно от вопросов их реализации. Более абстрактный стиль мышления повышает качество проектов.

Документация

Типы полезны также при *чтении* программ. Объявления типов в заголовках процедур и интерфейсах модулей — это разновидность документации, которая дает ценную информацию о поведении программы. Более того, в отличие от описаний, спрятанных в комментариях, такая документация не может устареть, поскольку она проверяется при каждом прогоне компилятора. Эта роль типов особенно существенна в сигнатурах модулей.

Безопасность языков

К сожалению, термин «безопасный язык» еще более расплывчат, чем «система типов». Как правило, программистам кажется, что они могут при взгляде на язык сказать, безопасен ли он. Однако, их мнение о том, что именно входит в понятие «безопасности языка», варьируется в зависимости от языкового сообщества, к которому они принадлежат. Неформально, впрочем, можно сказать, что безопасный язык ограждает программиста от совершения фатальных глупостей.

Немного уточняя интуитивное определение, можно сказать, что *безопасный язык — это язык, который защищает свои собственные абстракции*. Всякий язык высокого уровня предоставляет программисту абстрактный взгляд на работу машины. Безопасность означает, что язык способен гарантировать целостность этих абстракций, а также абстракций более высокого уровня, вводимых при помощи описательных средств языка. Например, язык может предоставлять массивы, с операциями доступа к ним и их изменения, абстрагируя нижележащую машинную память. Программист, использующий такой язык, ожидает, что массив можно модифицировать только путем явного использования операций обновления — а не, скажем, путем записи в память за границами какой-либо другой структуры данных. Аналогично можно ожидать, что к переменным со статической областью видимости можно обратиться только изнутри этой области, что стек вызовов действительно ведет себя как стек и т. д. В безопасном языке с такими абстракциями можно обращаться *абстрактно*, а в небезопасном — нельзя: в нем, чтобы полностью понять, как может повести себя программа, требуется держать в голове множество разнообразных низкоуровневых деталей, например, размещение структур данных в памяти и порядок их выделения компилятором. В предельном случае программы, написанные на небезопасных языках, могут поломать не только свои собственные структуры данных, но и структуры своей среды исполнения; при этом результаты могут быть совершенно непредсказуемыми.

Безопасность языка — это не то же самое, что статическая безопасность типов. Безопасности языка можно достичь при помощи системы типов, но можно добиться ее и с помощью проверок на стадии выполнения, которые отлавливают бессмысленные операции при попытке их исполнения, и останавливают программу или порождают исключение. К примеру, Scheme является безопасным языком, несмотря на отсутствие статической системы типов.

С другой стороны, в небезопасных языках часто имеются статические проверки типов, которые «по возможности» отлавливают хотя бы самые очевидные ошибки программистов. Впрочем, такие языки не могут считаться и безопасными с точки зрения типов, поскольку в общем случае они не могут гарантировать, что корректно типизированные программы ведут себя корректно — проверка типов в таком языке может предположить наличие ошибок во время выполнения (что, безусловно, лучше, чем ничего), но не доказать их отсутствие.

	Статическая проверка	Динамическая проверка
Безопасные	ML, Haskell, Java и т. п.	Lisp, Scheme, Perl, Postscript и т. п.
Небезопасные	C, C++ и т. п.	

Пустоту правой нижней клетки таблицы можно объяснить тем, что при наличии средств обеспечения безопасности при выполнении *большинства* операций нетрудно проверять их *все*. (На самом деле, существует несколько языков с динамической проверкой, например, некоторые диалекты Бейска для микроКомпьютеров с минимальными операционными системами, в которых есть низкоуровневые примитивы для чтения и записи произвольных ячеек памяти, при помощи которых можно нарушить целостность среды выполнения.)

Как правило, достичь безопасности выполнения при помощи только лишь статической типизации невозможно. Например, *все* языки, указанные в таблице как безопасные, на самом деле производят *проверку выхода за границы массивов* (array bounds checking) динамически.¹ Аналогично, языки со статической проверкой типов иногда предоставляют операции (скажем, *находящее преобразование типов* (down-casts) в Java — см. §15.5), некорректные с точки зрения проверки типов, а безопасность языка при этом достигается при помощи динамической проверки каждого употребления такой конструкции.

Безопасность языка редко бывает абсолютной. Безопасные языки часто предоставляют программистам «черные ходы», например, вызовы функций на других, возможно небезопасных, языках. Иногда такие контролируемые черные ходы даже содержатся в самом языке — например, `Obj.magic` в OCaml (Leroy, 2000), `Unsafe.cast` в Нью-Джерсийской реализации Standard ML, и т. п. Языки Modula-3 (Cardelli et al., 1989; Nelson, 1991) и C# (Wille, 2000) идут еще дальше и включают в себя «небезопасные подъязыки», предназначенные для реализации низкоуровневых библиотек вроде сборщиков мусора. Особые возможности этих подъязыков можно использовать только в модулях, явно помеченных словом `unsafe` («небезопасный»).

Карделли (Cardelli, 1996) смотрит на безопасность языка с другой точки зрения, проводя различие между *диагностируемыми* (trapped) и *недиагностируемыми* (untrapped) ошибками времени выполнения. Диагностируемая ошибка вызывает немедленную остановку вычисления (или порождает исключение, которое можно обработать внутри программы), в то время как при недиагностируемой ошибке вычисление может ещё продолжаться (по крайней мере, в течение некоторого времени). Пример недиагностируемой ошибки — обращение к данным за пределами массива в языке С. Безопасный язык, с этой точки зрения, — это язык, который предотвращает недиагностируемые ошибки во время выполнения.

Еще одна точка зрения основана на понятии переносимости; ее можно вы-

¹ Устранение проверок границ массива статическими средствами — хорошо известная цель при проектировании систем типов. В принципе, необходимые для этого механизмы (на основе *зависимых типов* (dependent types) — см. §30.5) хорошо изучены, однако их использование с соблюдением баланса между выразительной силой, предсказуемостью и разрешимостью проверки типов, а также сложностью программных аннотаций остается сложно задачей. О некоторых недавних достижениях в этой области можно прочитать у Си и Пфеннинга (Xi and Pfennig, 1998, 1999).

разить лозунгом «Безопасный язык полностью определяется руководством программиста». Сделаем так, чтобы программисту было достаточно понять *определение* (*definition*) языка, чтобы уметь предсказывать поведение любой программы на данном языке. Тогда руководство программиста на языке C не является его определением, поскольку поведение некоторых программ (скажем, тех, где встречается непроверенное обращение к массивам или используется арифметика указателей) невозможно предсказать, не зная, как конкретный компилятор C располагает структуры в памяти и т. п., а одна и та же программа может вести себя по-разному, будучи обработана разными компиляторами. Напротив, руководства по Java, Scheme и ML определяют (с различной степенью строгости) точное поведение любой программы, написанной на этих языках. Корректно типизированная программа получит одни и те же результаты в любой корректной реализации этих языков.

Эффективность

Первые системы типов в информатике появились в 50-х годах в таких языках, как Фортран ([Backus, 1981](#)), и были введены для того, чтобы повысить эффективность вычислений путем различения арифметических выражений с целыми и вещественными числами; это позволяло компилятору использовать различные представления чисел и генерировать соответствующие машинные команды для элементарных операций. В безопасных языках можно достичь большей эффективности, устранив многие динамические проверки, которые иначе потребовались бы для обеспечения безопасности (статически доказав, что проверка всегда даст положительный результат). В наше время большинство высокопроизводительных компиляторов существенным образом опираются при оптимизации и генерации кода на информацию, собранную процедурой проверки типов. Даже компиляторы для языков, в которых нет системы типов самой по себе, проделывают большую работу, чтобы хотя бы частично восстановить информацию о типах.

Информация о типах может принести выигрыш в эффективности в самых неожиданных местах. Например, недавно было показано, что с помощью информации, порождаемой при проверке типов, могут быть улучшены не только решения о генерации кода, но и представление указателей в параллелизованных научных вычислениях. Язык Titanium ([Yelick et al., 1998](#)) использует вывод типов для анализа области видимости указателей и способен принимать более эффективные решения на основе этих данных, чем программисты, оптимизирующие программы вручную (это подтверждается измерениями). Компилятор ML Kit с помощью мощного алгоритма *вывода регионов* (*region inference*) ([Gifford, Jouvelot, Lucassen, and Sheldon, 1987; Jouvelot and Gifford, 1991; Talpin and Jouvelot, 1992; Tofte and Talpin, 1994, 1997; Tofte and Birkedal, 1998](#)) заменяет большинство вызовов сборщика мусора (а иногда даже все вызовы) операциями управления памятью в стеке.

Другие приложения

Системы типов, помимо традиционных областей применения в программировании и проектировании языков, в последнее время используются в информатике и близких дисциплинах самыми разными способами. Очертим круг этих способов.

Все большее значение приобретает использование систем типов в области безопасности компьютеров и сетей. Например, статическая типизация лежит в основе модели безопасности Java и архитектуры автоматического конфигурирования (plug and play) сетевых устройств JINI (Arnold et al., 1999), а также играет ключевую роль в методике Proof Carrying Code («кода, содержащего доказательство», Necula and Lee, 1996, 1998; Necula, 1997). В то же время, многие фундаментальные идеи, возникшие в среде специалистов по безопасности, повторно используются в контексте языков программирования, и часто реализуются в виде системы анализа типов (напр., Abadi, Banerjee, Heintze, and Riecke, 1999; Abadi, 1999; Leroy and Rouaix, 1998; и т. д.). С другой стороны, растет интерес к прямому применению теории языков программирования в области компьютерной безопасности (напр., Abadi, 1999; Sumii and Pierce, 2001).

Алгоритмы проверки и вывода типов встречаются во многих инструментах анализа программ, помимо компиляторов. Например, утилита AnnoDomini, анализирующая программы на Коболе на предмет совместимости с проблемой 2000 года, построена на базе механизма вывода типов в стиле ML (Eidoff et al., 1999). Методы вывода типов использовались также в инструментах для анализа псевдонимов (pointer aliasing) (O'Callahan and Jackson, 1997) и анализа исключений (Leroy and Pessaux, 2000).

При автоматическом доказательстве теорем для представления логических утверждений и доказательств обычно используются очень мощные системы типов, основанные на зависимых типах. Некоторые популярные средства работы с доказательствами, включая Nuprl (Constable et al., 1986), Lego (Luo and Pollack, 1992; Pollack, 1994), Coq (Barras, Boutin, Cornes, Courant, Filliatre, Gimenez, Herbelin, Huet, Munoz, Murthy, Parent, Paulin-Mohring, Saibi, and Werner, 1997) и Alf (Magnusson and Nordström, 1994), прямо основаны на теории типов. Констебль (Constable, 1998) и Пфенниг (Pfenning, 1999) излагают в своих работах историю этих систем.

Растет интерес к системам типов и в сообществе специалистов по базам данных. Это связано с популярностью «сетевых метаданных», использующихся для описания структурированных данных на XML, таких как DTD (Document Type Definitions, XML 1998) и других видов схем (таких, как новый стандарт XML-Schema, XS 2000). Новые языки для запросов к XML и обработки XML-данных обладают мощными статическими системами типов, прямо основанными на этих языках схем (Hosoya and Pierce, 2000; Hosoya, Vouillon, and Pierce, 2001; Hosoya and Pierce, 2001; Relax, 2000; Shields, 2001).

Совершенно отдельная область приложения систем типов — вычислительная лингвистика, где типизированные лямбда-исчисления лежат в основе таких формализмов, как *категориальная грамматика* (categorial grammar) (van Benthem, 1995; van Benthem and Meulen, 1997; Ranta, 1995, и т. д.).

1.3. Системы типов и проектирование языков

Встраивание системы типов в существующий язык, в котором она не была предусмотрена изначально, может оказаться непростой задачей; в идеале проектирование языка должно идти одновременно с проектированием системы типов.

Одна из причин этого в том, что языки без систем типов — даже безопасные языки с динамическими проверками, — как правило, содержат такие конструкции или рекомендуют такие идиоматические приемы, которые осложняют проверку типов или даже делают ее невозможной. В типизированных же языках система типов сама по себе часто рассматривается как фундамент проекта языка, как его организующий принцип, в свете которого оцениваются все проектные решения разработчика.

Еще один фактор заключается в том, что, как правило, синтаксис типизированных языков сложнее, чем у нетипизированных, поскольку приходится поддерживать объявления типов. Построить красивый и понятный синтаксис проще, если сразу учитывать все эти вопросы.

Утверждение, что типы должны являться неотъемлемой частью языка программирования, ортогонально вопросу о том, в каких случаях программист должен явно указывать аннотации типа, а в каких случаях их может вычислить компилятор. Хорошо спроектированный статически типизированный язык никогда не будет требовать от программиста вручную выписывать длинные и утомительные декларации типов. Впрочем, по вопросу о том, какое количество явной информации считать чрезмерным, существуют различные мнения. Создатели языков семейства ML тщательно старались свести аннотации к самому минимуму, используя для получения недостающей информации методы вывода типов. В языках семейства C, включая Java, принят несколько более многословный стиль.

1.4. Краткая история

Самые ранние системы типов в информатике использовались для простейшего различия между целыми числами и числами с плавающей точкой (например, в Фортране). В конце 1950-х и начале 1960-х эта классификация была расширена на структурированные данные (массивы записей и т. п.) и функции высшего порядка. В начале 1970-х появились еще более сложные понятия (параметрический полиморфизм, абстрактные типы данных, системы модулей и подтипы), и системы типов превратились в отдельное направление исследований. Тогда же специалисты по информатике обнаружили связь между системами типов в языках программирования и типами, изучаемыми в математической логике, и это привело к плодотворному взаимодействию между двумя областями, которое продолжается по сей день.

На рис. 1.1 представлена краткая (и чрезвычайно неполная!) хронология основных достижений в истории систем типов в информатике. Курсивом отмечены открытия в области логики, чтобы показать важность достижений в этой области. Ссылки на работы, указанные в правой колонке, можно найти

в библиографии.

1.5. Дополнительная литература

Хотя я пытался сделать эту книгу самодостаточной, она далека от всеохватывающей; область исследований теории типов слишком широка, и говорить о ней можно с разных точек зрения, поэтому все ее аспекты невозможно достойно осветить в одной книге. В этом разделе я укажу некоторые другие вводные тексты.

В обзорных статьях Карделли ([Cardelli, 1996](#)) и Митчелла ([Mitchell, 1990b](#)) содержится краткое введение в дисциплину. Статья Барендргета ([Barendregt, 1992](#)) предназначена скорее для читателя, склонного к математике. Объемистый учебник Митчелла «Основания языков программирования» ([«Foundations for Programming Languages», Mitchell, 1996](#)) описывает основы лямбда-исчисления, несколько систем типов и многие вопросы семантики. Основное внимание уделяется семантике, а не деталям реализации. Книга Рейнольдса «Теории языков программирования» ([«Theories of Programming Languages», Reynolds, 1998b](#)) — это обзор теории языков программирования, предназначенный для аспирантов и содержащий изящное описание полиморфизма, подтипов и типов-пересечений. «Структура типизированных языков программирования» Шмидта ([«The Structure of Typed Programming Languages», Schmidt, 1994](#)) развивает основные понятия систем типов в контексте проектирования языков, и содержит несколько глав по обычновенным императивным языкам программирования. Монография Хиндли «Основы теории простых типов» ([«Basic Simple Type Theory», Hindley, 1997](#)) является замечательным собранием результатов теории простого типизированного лямбда-исчисления и близкородственных систем. Оно отличается скорее глубиной, нежели широтой охвата.

«Теория объектов» Абади и Карделли ([«A Theory of Objects», Abadi and Cardelli, 1996](#)) развивает во многом тот же материал, что и настоящая книга, с меньшим упором на вопросы реализации. Вместо этого она подробно описывает применение основных идей для построения оснований теории объектно-ориентированного программирования. Книга Кима Брюса «Основы объектно-ориентированных языков: типы и семантика» ([«Foundations of Object-Oriented Languages: Types and Semantics», Bruce, 2002](#)) покрывает приблизительно тот же материал. Введение в теорию объектно-ориентированных систем типов можно также найти в книгах Палсберга и Шварцбаха ([Palsberg and Schwartzbach, 1994](#)), а также Кастаньи ([Castagna, 1997](#)).

Семантические основы как бестиповых, так и типизированных языков подробно рассмотрены в учебниках Гантера ([Gunter, 1992](#)), Уинскеля ([Winskel, 1993](#)) и Митчелла ([Mitchell, 1996](#)). Кроме того, операционная семантика детально описана в книге Хенnessи ([Hennessy, 1990](#)). Основания семантики типов в рамках математической теории категорий (category theory) можно найти во множестве источников, включая книги Якобса ([Jacobs, 1999](#)), Асперти и Лонго ([Asperti and Longo, 1991](#)) и Кроула ([Crole, 1994](#)); краткое введение имеется в «Основах теории категорий для специалистов по информатике» ([«Basic](#)

Category Theory for Computer Scientists», Pierce, 1991a).

Книга Жирара, Лафонта и Тейлора «Доказательства и типы» («Proofs and Types», Girard, Lafont, and Taylor, 1989) посвящена логическим вопросам теории типов (изоморфизм Карри-Говарда и т. п.). Кроме того, она включает описание Системы F , сделанное ее создателем, и приложение с введением в линейную логику. Связи между типами и логикой исследуются также в книге «Вычисление и дедукция» Пфеннинга («Computation and Deduction», Pfennig, 2001). «Теория типов и функциональное программирование» Томпсона («Type Theory and Functional Programming», Thompson, 1991) и «Конструктивные основания функциональных языков» Тёрнера («Constructive Foundations for Functional Languages», Turner, 1991) посвящены связям между функциональным программированием (в смысле «чисто функционального программирования», как в языках Haskell и Miranda) и конструктивной теорией типов, рассматриваемой с точки зрения логики. Множество вопросов теории доказательств, имеющих отношение к программированию, рассмотрены в книге «Теория типов и автоматическая дедукция» Гобольт-Ларреクса и Макки («Proof Theory and Automated Deduction», Goubault-Larrecq and Mackie, 1997). История типов в логике и философии более подробно описана в статьях Констебля (Constable, 1998), Уодлера (Wadler, 2000), Юэ (Huet, 1990) и Пфеннинга (Pfenning, 1999), а также в диссертации Лаана (Laan, 1997) и в книгах Граттан-Гиннесса (Grattan-Guinness, 2001) и Соммаруги (Sommaruga, 2000).

Как выясняется, чтобы избежать досадных ошибочных утверждений о корректности типов в языках программирования, требуется немалая доля тщательного анализа. Вследствие этого классификация, описание и исследование систем типов превратились в формальную дисциплину.

Лука Карделли (1996)

1870-е	<i>начала формальной логики</i>	Frege (1879)
1900-е	<i>формализация математики</i>	Whitehead and Russell (1910)
1930-е	<i>бестиповое лямбда-исчисление</i>	Church (1941)
1940-е	<i>простое типизированное лямбда-исчисление</i>	Church (1940), Curry and Feys (1958)
1950-е	Фортран Алгол-60	Backus (1981) Naur et al. (1963)
1960-е	<i>проект Automath</i> Симула	de Bruijn (1980) Birtwistle, Dahl, Myhrhaug, and Nygaard (1979) Howard (1980) van Wijngaarden, Mailloux, Peck, Koster, Sintzoff, Lindsey, Meertens, and Fisker (1975) Wirth (1971)
	<i>изоморфизм Карри-Говарда</i> Алгол-68	Martin-Löf (1973, 1982) Girard (1972) Reynolds (1974)
1970-е	Паскаль <i>Теория типов Мартина-Лёфа</i> <i>Системы F, F_ω</i> полиморфное лямбда-исчисление CLU	Liskov, Atkinson, Bloom, Moss, Schaffert, Scheifler, and Snyder (1981) Milner (1978), Damas and Milner (1982) Gordon, Milner, and Wadsworth (1979)
	полиморфный вывод типов	Coppo and Dezani-Ciancaglini (1978) Coppo, Dezani-Ciancaglini, and Sallé (1979), Pottinger (1980)
	ML	Constable et al. (1986)
	<i>типы-пересечения</i>	Reynolds (1980), Cardelli (1984), Mitchell (1984a)
1980-е	проект Nuprl подтипы	Mitchell and Plotkin (1988)
	АТД как экзистенциальные типы <i>исчисление конструкций</i>	Coquand (1985), Coquand and Huet (1988)
	<i>линейная логика</i>	Girard (1987), Girard, Lafont, and Taylor (1989)
	ограниченная квантификация	Cardelli and Wegner (1985)
	<i>LF (Edinburgh Logical Framework)</i> Forsythe	Curien and Ghelli (1992), Cardelli, Martini, Mitchell, and Scedrov (1994)
	<i>чистые системы типов</i>	Harper, Honsell, and Plotkin (1992)
	зависимые типы и модульность Quest	Reynolds (1988)
	системы эффектов	Terlouw (1989), Berardi (1988), Barendregt (1991)
	строчные переменные; расширяющие записи	MacQueen (1986)
<i>-sourcefile-</i>	мые записи	Cardelli (1991)
1990-е	подтипы высших порядков	Gifford, Jouvelot, Lucassen, and Sheldon (1987), Talpin and Jouvelot (1992)
		Wand (1987), Rémy (1989)
		Rev: -revision-, November 28, 2010
		Cardelli and Mitchell (1991)
		Cardelli (1990), Cardelli and Longo (1991)

Глава 2

Математический аппарат

Прежде чем начать, нам нужно договориться об используемых обозначениях и сообщить несколько базовых математических утверждений. Большинство читателей может бегло проглядеть эту главу и затем возвращаться к ней по мере необходимости.

2.1. Множества, отношения и функции

Определение 2.1.1 Мы пользуемся стандартными обозначениями для множеств: фигурные скобки используются, когда элементы множества перечисляются явным образом ($\{\dots\}$) или когда оно задается выделением (*comprehension*) из другого множества ($\{x \in S \mid \dots\}$), \emptyset обозначает пустое множество, а выражение $S \setminus T$ — теоретико-множественную разность S и T (множество элементов S , не являющихся элементами T). Мощность (количество элементов) множества S обозначается $|S|$. Множество всех подмножеств S обозначается $\mathcal{P}(S)$.

Определение 2.1.2 Множество натуральных чисел (*natural numbers*) $\{0, 1, 2, 3, 4, 5, \dots\}$ обозначается символом \mathbb{N} . Множество называется счетным (*countable*), если между его элементами и натуральными числами существует взаимно-однозначное соответствие.

Определение 2.1.3 n -местное отношение (*relation*) на наборе множеств S_1, S_2, \dots, S_n — это множество $R \subseteq S_1 \times S_2 \times \dots \times S_n$ кортежей элементов S_1, \dots, S_n . Если $(s_1, \dots, s_n) \in R$, где $s_1 \in S_1, \dots, s_n \in S_n$, то говорится, что s_1, \dots, s_n связаны (*related*) отношением R .

Определение 2.1.4 Одноместное отношение на множестве S называется предикатом (*predicate*) на S . Говорится, что P истинен на $s \in S$, если $s \in P$. Чтобы подчеркнуть это интуитивное понятие, мы часто будем писать $P(s)$ вместо $s \in P$, рассматривая P как функцию, переводящую элементы S в истинностные значения.

Определение 2.1.5 Двуместное отношение R на множествах S и T называется бинарным отношением (*binary relation*). Мы часто будем писать $s R t$ вместо $(s, t) \in R$. Если S и T совпадают (назовем это множество U), мы будем говорить, что R — бинарное отношение на U .

Определение 2.1.6 Чтобы облегчить чтение, многоместные (трех- и более) отношения часто записываются в «смешанном» синтаксисе, когда элементы отношения разделяются последовательностью символов, и эти символы вместе образуют имя отношения. Например, отношение типизации для простого типизированного лямбда-исчисления из главы 9 записывается как $\Gamma \vdash s : T$ — такая формула означает «тройка (Γ, s, T) находится в отношении типизации».

Определение 2.1.7 Областью определения (*domain*) отношения R между множествами S и T называется множество элементов $s \in S$, таких, что $(s, t) \in R$ для некоторого t . Область определения R обозначается $\text{dom}(R)$. Областью значений (*codomain*) R называется множество элементов $t \in T$, таких, что $(s, t) \in R$ для некоторого s . Область значений R обозначается $\text{range}(R)$.

Определение 2.1.8 Отношение R на множествах S и T называется частичной функцией (*partial function*) из S в T , если из $(s, t_1) \in R$ и $(s, t_2) \in R$ следует $t_1 = t_2$. Если, кроме того, $\text{dom}(R) = S$, то R называется всюду определенной функцией (*total function*), или просто функцией (*function*), из S в T .

Определение 2.1.9 Частичная функция R из S в T определена (*defined*) на аргументе $s \in S$, если $s \in \text{dom}(R)$, и не определена в противном случае. Запись $f(x) \uparrow$ или $f(x) = \uparrow$ означает « f не определена на x », а $f(x) \downarrow$ означает « f определена на x ».

В некоторых главах, посвященных программной реализации систем типов, нам также потребуется определять функции, которые на некоторых аргументах терпят неудачу (*fail*) (см., например, рис. 22.2). Важно отличать неудачу (частный случай разрешенного, наблюдаемого результата) от расхождения (*divergence*); функция, способная потерпеть неудачу, может быть либо частичной (т. е., может также расходиться), либо быть всюду определенной (т. е., она всегда либо возвращает результат, либо терпит неудачу) — в сущности, часто нам будет нужно доказывать, что такая функция всюду определена. Если f возвращает неудачу при применении к x , мы будем писать $f(x) = \text{неудача}$.

С формальной точки зрения, функция из S в T , которая может вернуть неудачу, является на самом деле функцией из S в $T \cup \{\text{неудача}\}$ (мы предполагаем, что неудача не входит во множество T).

Определение 2.1.10 Пусть R — бинарное отношение на множестве S , а P — предикат на S . Если из $s R s'$ и $P(s)$ следует $P(s')$, то говорится, что R сохраняет (*preserves*) P .

2.2. Упорядоченные множества

Определение 2.2.1 Бинарное отношение R на множестве S рефлексивно (*reflexive*), если каждый элемент S связан отношением R с самим собой — то есть, для всех $s \in S$, $s R s$ (или $(s, s) \in R$). Отношение R симметрично (*symmetric*), если для всех $s, t \in S$ из $s R t$ следует $t R s$. Отношение R транзитивно (*transitive*), если из $s R t$ и $t R u$ следует $s R u$. Отношение R антисимметрично (*antisymmetric*), если из $s R t$ и $t R s$ следует $s = t$.

Определение 2.2.2 Рефлексивное и транзитивное отношение R на множестве S называется предпорядком (*preorder*) на S . (Всякий раз, когда мы говорим о «предупорядоченном множестве S », мы имеем в виду какой-то конкретный предпорядок на S .) Предпорядки обычно обозначаются символами \leq или \sqsubseteq . Запись $s < t$ означает, что $s \leq t \wedge s \neq t$ (« s строго меньше t »).

Если предпорядок (на S) к тому же антисимметричен, он называется частичным порядком (*partial order*) на S . Частичный порядок \leq называется линейным порядком (*total order*) на S , если для любых $s, t \in S$ выполняется либо $s \leq t$, либо $t \leq s$.

Определение 2.2.3 Пусть \leq — частичный порядок на S , а s и t — элементы S . Элемент $j \in S$ называется объединением (*join*) (или точной верхней границей, *least upper bound*) s и t , если:

1. $s \leq j$ и $t \leq j$, а также
2. для всякого $k \in S$, если $s \leq k$ и $t \leq k$, то $j \leq k$.

Аналогично, элемент $m \in S$ называется пересечением (*meet*) (или точной нижней границей, *greatest lower bound*) s и t , если:

1. $m \leq s$ и $m \leq t$, а также
2. для всякого $n \in S$, если $n \leq s$ и $n \leq t$, то $n \leq m$.

Определение 2.2.4 Рефлексивное, транзитивное и симметричное отношение на множестве S называется отношением эквивалентности (*equivalence relation*) на S .

Определение 2.2.5 Пусть R — бинарное отношение на множестве S . Рефлексивное замыкание (*reflexive closure*) R — это наименьшее рефлексивное отношение R' , содержащее в себе R . («Наименьшее» здесь означает, что, если имеется какое-то другое рефлексивное отношение R'' , включающее все пары из R , то $R' \subseteq R''$.) Аналогично, транзитивное замыкание (*transitive closure*) R — это наименьшее транзитивное отношение R' , содержащее R . Транзитивное замыкание R часто обозначается R^+ . Рефлексивно-транзитивное замыкание (*reflexive and transitive closure*) R — это наименьшее рефлексивное и транзитивное отношение, содержащее R . Оно часто обозначается R^* .

Упражнение 2.2.6 [★★ →] Дано отношение R на множестве S . Определим отношение R' так:

$$R' = R \cup \{(s, s) \mid s \in S\}$$

То есть, R' содержит все пары из R плюс все пары вида (s, s) . Покажите, что R' является рефлексивным замыканием R .

Упражнение 2.2.7 [★★ →] Вот более конструктивный способ определения транзитивного замыкания отношения R . Сначала определим следующую последовательность множеств пар:

$$\begin{aligned} R_0 &= R \\ R_{i+1} &= R_i \cup \{(s, u) \mid \text{для некоторого } t, (s, t) \in R_i \text{ и } (t, u) \in R_i\} \end{aligned}$$

То есть, на каждом шаге $i + 1$ мы добавляем к R_i все пары, которые можно получить «за один шаг транзитивности» из пар, входящих в R_i . Наконец, определим отношение R^+ как объединение всех R_i :

$$R^+ = \bigcup_i R_i$$

Покажите, что R^+ на самом деле является транзитивным замыканием R — т. е., что оно удовлетворяет условиям, заданным в определении 2.2.5.

Упражнение 2.2.8 [★★ →] Пусть R — бинарное отношение на множестве S , и это отношение сохраняет предикат P . Докажите, что R^* также сохраняет P .

Определение 2.2.9 Пусть у нас есть предпорядок \leqslant на множестве S . Убывающая цепочка (*decreasing chain*) на \leqslant есть последовательность s_1, s_2, s_3, \dots элементов S , такая, что каждый элемент последовательности строго меньше предшествующего: $s_{i+1} < s_i$ для всякого i . (Цепочки могут быть конечными или бесконечными, однако для нас представляют больший интерес бесконечные, как видно из следующего определения.)

Определение 2.2.10 Пусть у нас есть множество S с предпорядком \leqslant . Предпорядок \leqslant называется полным (*well founded*), если он не содержит бесконечных убывающих цепочек. Например, обычный порядок на натуральных числах, $0 < 1 < 2 < 3 < \dots$, является полным, а тот же самый порядок на целых числах, $\dots < -3 < -2 < -1 < 0 < 1 < 2 < 3 < \dots$ — нет. Иногда мы не упоминаем \leqslant явно и просто называем S вполне упорядоченным множеством (*well-founded set*).

2.3. Последовательности

Определение 2.3.1 Последовательность (*sequence*) записывается путем перечисления элементов через запятую. Запятая используется как для добавления элемента в начало (аналогично операции *cons* в языке *Lisp*) или конец последовательности, так и для склеивания двух последовательностей

(аналогично *append* в *Lisp*). Например, если символ *a* обозначает последовательность 3, 2, 1, а символ *b* обозначает последовательность 5, 6, то 0, *a* — это последовательность 0, 3, 2, 1, запись *a*, 0 обозначает последовательность 3, 2, 1, 0, а запись *b*, *a* обозначает 5, 6, 3, 2, 1. (Использование запятой для двух разных операций — аналогов *cons* и *append*, — не создает путаницы, если нам не нужно вести речь о последовательностях последовательностей.) Последовательность чисел от 1 до *n* обозначается 1..*n* (через две точки). Запись |*a*| означает длину последовательности *a*. Пустая последовательность обозначается знаком • или пробелом. Одна последовательность называется перестановкой (*permutation*) другой последовательности, если они содержат в точности одни и те же элементы, которые могут быть расположены в них в разном порядке.

2.4. Индукция

Доказательства по индукции встречаются в теории языков программирования очень часто, как и в большинстве разделов информатики. Многие из этих доказательств основаны на одном из следующих принципов:

Аксиома 2.4.1 [ПРИНЦИП ОБЫКНОВЕННОЙ ИНДУКЦИИ НА НАТУРАЛЬНЫХ ЧИСЛАХ] Пусть *P* — предикат, заданный на множестве натуральных чисел. Тогда

Если $P(0)$
и для любого *i*, из $P(i)$ следует $P(i + 1)$,
то $P(n)$ выполняется для всех *n*.

Аксиома 2.4.2 [ПРИНЦИП ПОЛНОЙ ИНДУКЦИИ НА НАТУРАЛЬНЫХ ЧИСЛАХ] Пусть *P* — предикат на множестве натуральных чисел. Тогда

Если для каждого натурального числа *n*,
предполагая, что $P(i)$ для всех *i* $< n$,
мы можем показать, что $P(n)$,
то $P(n)$ выполняется для всех *n*.

Определение 2.4.3 Лексикографический (или «словарный») порядок (*lexicographic order*) на парах натуральных чисел определяется следующим образом: $(m, n) \leqslant (m', n')$ тогда и только тогда, когда либо $m < m'$, либо $m = m'$ и $n \leqslant n'$.

Аксиома 2.4.4 [ПРИНЦИП ЛЕКСИКОГРАФИЧЕСКОЙ ИНДУКЦИИ] Пусть *P* — предикат на множестве пар натуральных чисел. Тогда

Если для каждой пары натуральных чисел (m, n) ,
предполагая, что $P(m', n')$ для всех $(m', n') < (m, n)$,
мы можем показать, что $P(m, n)$,

то $P(m, n)$ выполняется для всех m, n .

Принцип лексикографической индукции служит основой для доказательств с *вложенной индукцией* (inner induction), когда какой-либо пункт индуктивного доказательства использует индукцию «внутри себя». Этот принцип можно распространить на индукцию по тройкам, четверкам и т. д. (Индукция по парам нужна достаточно часто; индукция по тройкам иногда может быть полезной; индукция по четверкам и далее встречается редко.)

В теореме 3.3.4 вводится еще один вариант доказательств по индукции, называемый *структурной индукцией* (structural induction). Он особенно полезен для доказательства утверждений о древовидных структурах вроде термов или деревьях вывода типов. Математические основания индуктивных рассуждений будут подробнее рассмотрены в главе 21. Мы увидим, что все упомянутые принципы индукции являются частными проявлениями единой, более общей идеи.

2.5. Справочная литература

Тем, кому понятия, перечисленные в этой главе, оказались незнакомыми, вероятно, имеет смысл ознакомиться со справочной литературой. Существует множество вводных курсов. В частности, книга Винскела (1993) помогает развить интуицию в вопросах индукции. Начальные главы в книге Дэйви и Пристли (1990) содержат замечательный обзор по упорядоченным множествам. Хэлмос (1987) служит хорошим введением в элементарную теорию множеств.

Доказательство есть
воспроизводимый опыт в деле
убеждения.

Джим Хорнинг

Часть I

Бестиповые системы

Глава 3

Бестиповые арифметические выражения

Для корректного обсуждения систем типов и их свойств мы прежде всего должны научиться формально обращаться с некоторыми базовыми характеристиками языков программирования. В частности, нам требуются четкие, ясные и математически точные механизмы, позволяющие описывать синтаксис и семантику программ, а также строить рассуждения о них.

В этой и последующей главах мы разработаем такие механизмы для небольшого языка, содержащего числа и логические значения. Сам этот язык настолько тривиален, что почти не заслуживает рассмотрения, однако с его помощью мы вводим некоторые базовые понятия — абстрактный синтаксис, индуктивные определения и доказательства, вычисление, а также моделирование ошибок времени выполнения. В главах с 5 по 7 те же самые шаги проделываются для намного более мощного языка: бестипового лямбда-исчисления, в котором нам также приходится иметь дело со связыванием имен и подстановками. Далее, в главе 8, мы начинаем собственно изучение типов, возвращаясь к простому языку из этой главы и вводя с его помощью основные понятия статической типизации. В главе 9 эти понятия будут применены к лямбда-исчислению.

3.1. Введение

В языке этой главы имеется лишь несколько синтаксических конструкций: булевские константы `true` (истина) и `false` (ложь), условные выражения, числовая константа 0, арифметические операторы `succ` (следующее число) и `pred` (предыдущее число) и операция проверки `iszzero`, которая возвращает значение

В этой главе изучается бестиповое исчисление логических значений и чисел (рис. 3.2 на с. 60). Соответствующая реализация на OCaml хранится в веб-репозитории под именем `arith` и описывается в главе 4. Инструкции по скачиванию и сборке программы проверки можно найти по адресу <http://www.cis.upenn.edu/~bcpierce/tapl>.

ние `true`, будучи применена к 0, и значение `false`, когда применяется к любому другому числу. Эти конструкции можно компактно описать следующей грамматикой.

<code>t ::=</code>	<i>термы:</i>
<code>true</code>	константа «истина»
<code>false</code>	константа «ложь»
<code>if t then t else t</code>	условное выражение
<code>0</code>	константа «ноль»
<code>succ t</code>	следующее число
<code>pred t</code>	предыдущее число
<code>iszzero t</code>	проверка на ноль

Формат описания этой грамматики (и других грамматик во всем тексте этой книги) близок к стандартной форме Бэкуса-Наура (см. Aho, Sethi, and Ullman, 1986). В первой строке (`t ::=`) определяется набор *термов* (terms), а также объявляется, что для обозначения термов мы будем употреблять букву `t`. Остальные строки описывают синтаксические формы, допустимые для термов. Всюду, где встречается символ `t`, можно подставить любой терм. В правой колонке курсивом набраны комментарии.

Символ `t` в правой части описания грамматики называется *метапеременной* (metavariable). Это переменная в том смысле, что `t` служит в качестве заместителя какого-то конкретного терма, но это «мета»-переменная, поскольку она не является *переменной объектного языка* (object language) — то есть, самого языка программирования, синтаксис которого мы описываем, — а *метаязыка* (metalanguage), знаковой системы, используемой для описания. (На самом деле, в этом объектном языке даже нет переменных; мы их введем лишь в главе 5.) Приставка *мета*- происходит из *метаматематики* (metamathematics) — отрасли логики, которая изучает математические свойства систем, предназначенных для математических и логических рассуждений (в частности, языков программирования). Оттуда же происходит термин *метатеория* (metatheory); он означает совокупность истинных утверждений, которые мы можем сделать о какой-либо логической системе (или о языке программирования), а в расширенном смысле — исследование таких утверждений. То есть, такое выражение как «метатеория подтипов» в этой книге может пониматься как «формальное исследование свойств систем с подтипами».

В тексте этой книги мы будем использовать метапеременную `t`, а также соседние буквы, скажем, `s`, `u` и `r`, и варианты вроде `t1` или `s'`, для обозначения термов того объектного языка, которым мы занимаемся в данный момент; далее будут введены и другие буквы для обозначения выражений других синтаксических категорий. Полный список соглашений по использованию метапеременных можно найти в приложении B.

Пока что слова *терм* и *выражение* обозначают одно и то же. Начиная с главы 8, в которой мы станем изучать исчисления, обладающие более богатым набором синтаксических категорий (таких, как *типы*), словом *выражение* мы будем называть любые синтаксические объекты (в том числе выражения-термы, выражения-типы, выражения-виды и т. п.), а *терм* будет употребляться в более узком смысле — обозначая выражения, которые представляют

собой вычисления (т. е. такие выражения, которые можно подставить вместо метапеременной `t`).

Программа на нашем нынешнем языке — это всего лишь терм, построенный при помощи перечисленных выше конструкций. Вот пара примеров программ и результат их вычисления. Для краткости мы используем стандартные арабские цифры для записи чисел, которые формально представляются как набор последовательных применений операции `succ` к 0. Например, `succ(succ(succ(0)))` записывается как 3.

```
if false then 0 else 1;

⇒ 1

iszero (pred (succ 0));

⇒ true
```

Символом `⇒` в тексте книги обозначаются результаты вычисления примеров. (Для краткости результаты опускаются, когда они очевидны или не играют никакой роли.) При верстке результаты автоматически вставляются реализацией той формальной системы, которая рассматривается в данный момент (в этой главе — `arith`); напечатанные результаты представляют собой реальный вывод приложения.*

Составные аргументы `succ`, `pred` и `iszero` в примерах приведены, для удобства чтения, в скобках.¹ Скобки не упоминаются в грамматике термов; она определяет только *абстрактный синтаксис* (abstract syntax). Разумеется, присутствие скобок или их отсутствие играет весьма малую роль в чрезвычайно простом языке, с которым мы сейчас имеем дело: обычно скобки служат для разрешения неоднозначностей в грамматике, но в нашей грамматике неоднозначностей нет — любая последовательность символов может быть интерпретирована как терм максимум одним способом. Мы вернемся к обсуждению скобок и абстрактного синтаксиса в главе 5 (с. 73).

Результатом вычислений служат термы самого простого вида: это всегда либо булевские константы, либо числа (последовательные вызовы `succ` ноль или более раз с аргументом 0). Такие термы называются *значениями* (values), и они будут играть особую роль при формализации порядка вычисления термов.

Заметим, что синтаксис термов позволяет образовывать термы сомнительного вида, вроде `succ true` или `if 0 then 0 else 0`. Мы поговорим о таких термах позднее — в сущности, именно их наличие делает этот крошечный язык интересным для наших целей, поскольку они являются примерами именно тех бессмысленных программ, которых мы хотим избежать при помощи системы типов.

* При верстке перевода это правило не соблюдалось. — *прим. перев.*

¹ На самом деле интерпретатор, которым обрабатывались примеры из этой главы (на сайте книги он называется `arith`) требует скобок вокруг составных аргументов `succ`, `pred` и `iszero`, даже несмотря на то, что их можно однозначно разобрать и без скобок. Это сделано для совместимости с последующими исчислениями, где подобный синтаксис используется при применении функций к аргументам.

3.2. Синтаксис

Есть несколько эквивалентных способов определить синтаксис нашего языка. Один из них мы уже видели — это грамматика, приведенная на с. 44. Грамматика эта, в сущности, всего лишь сокращенная форма записи следующего индуктивного определения:

Определение 3.2.1 [ТЕРМЫ, индуктивно]: *Множество термов — это наименьшее множество \mathcal{T} такое, что:*

1. $\{\text{true}, \text{false}, 0\} \subseteq \mathcal{T}$;
2. если $t_1 \in \mathcal{T}$, то $\{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\} \subseteq \mathcal{T}$;
3. если $t_1 \in \mathcal{T}$, $t_2 \in \mathcal{T}$, $t_3 \in \mathcal{T}$, то $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}$.

Такие индуктивные определения повсеместно встречаются при исследовании языков программирования, так что имеет смысл остановиться и рассмотреть наше определение подробнее. Первый пункт задаёт три простых выражения, содержащихся в \mathcal{T} . Второй и третий пункты указывают правила, с помощью которых мы можем выяснить, содержатся ли некоторые сложные выражения в \mathcal{T} . Наконец, слово «наименьшее» говорит, что в \mathcal{T} нет никаких элементов, кроме требуемых этими тремя пунктами.

Как и грамматика, приведенная на с. 44, это определение ничего не говорит об использовании скобок для выделения составных подтермов. С формальной точки зрения, мы определяем \mathcal{T} как множество деревьев, а не множество строк. Скобки в примерах используются для того, чтобы показать, как линейная форма термов, которую мы записываем на бумаге, соотносится с внутренней древовидной формой.

Еще один способ представления того же индуктивного определения термов использует двумерную запись в виде *правил вывода* (inference rules), часто используемую в форме «естественного вывода» для представления логических систем:

Определение 3.2.2 [ТЕРМЫ, ЧЕРЕЗ ПРАВИЛА ВЫВОДА]: *Множество термов определяется следующими правилами:*

$$\begin{array}{c}
 \text{true} \in \mathcal{T} \qquad \qquad \text{false} \in \mathcal{T} \qquad \qquad 0 \in \mathcal{T} \\
 \\
 \frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \qquad \qquad \frac{t_1 \in \mathcal{T}}{\text{pred } t_1 \in \mathcal{T}} \qquad \qquad \frac{t_1 \in \mathcal{T}}{\text{iszero } t_1 \in \mathcal{T}} \\
 \\
 \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}}
 \end{array}$$

Первые три правила повторяют первый пункт определения 3.2.1; остальные четыре соответствуют пунктам (2) и (3). Каждое правило вывода читается так: «Если мы установили истинность предпосылок, указанных над чертой, то мы можем прийти к заключению под чертой». Утверждение о том, что \mathcal{T}

должно являться *наименьшим* множеством, удовлетворяющим этим правилам, зачастую не приводится явно (как в данном случае).

Стоит упомянуть два терминологических момента. Во-первых, правила без предпосылок (как первые три в нашем определении) часто называют *аксиомами* (axioms). В этой книге термин *правило вывода* используется как для «собственно правил вывода», для которых имеется одна или несколько предпосылок, так и для аксиом. Аксиомы обычно записываются без черты, поскольку над ней нечего писать. Во-вторых, если выражаться абсолютно точно, наши «правила вывода» на самом деле представляют собой *схемы правил* (rule schemas), поскольку предпосылки и заключения в них могут содержать метапеременные. С формальной точки зрения, каждая схема представляет бесконечное множество *конкретных правил* (concrete rules), получаемых путем замены каждой метапеременной различными выражениями, которые принадлежат соответствующей синтаксической категории — т. е., в данном случае, вместо каждого t подставляются все возможные термы.

Наконец, то же самое множество термов можно определить еще одним способом, в более «конкретном» стиле, явно указав процедуру *порождения* (generation) элементов \mathcal{T} .

Определение 3.2.3 [ТЕРМЫ, КОНКРЕТНЫМ ОБРАЗОМ]: Для каждого натурального числа i определим множество S_i :

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= \{ \text{true}, \text{false}, 0 \} \\ &\cup \{ \text{succ } t_1, \text{pred } t_1, \text{iszzero } t_1 \mid t_1 \in S_i \} \\ &\cup \{ \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i \}. \end{aligned}$$

Наконец, пусть

$$S = \bigcup_i S_i.$$

S_0 пусто; S_1 содержит только константы; S_2 содержит константы и выражения, которые можно построить из констант путем применения одной из операций `succ`, `pred`, `iszzero` или `if`; S_3 содержит все эти выражения плюс те, которые можно построить за одно применение `succ`, `pred`, `iszzero` или `if` к элементам S_2 ; и так далее. S собирает вместе все эти выражения — т. е., все выражения, которые можно получить применением конечного числа арифметических и условных операторов, начиная с констант.

Упражнение 3.2.4 [★★]: Сколько элементов содержит S_3 ?

Упражнение 3.2.5 [★★]: Покажите, что множества S_i кумулятивны (cumulative) — то есть, для любого i выполняется $S_i \subseteq S_{i+1}$.

Рассмотренные нами определения характеризуют одно и то же множество термов с разных точек зрения: определения 3.2.1 и 3.2.2 просто описывают множество как наименьшее, имеющее некоторые «свойства замыкания»; определение 3.2.3 показывает, как построить множество в виде предела последовательности.

Завершая наше обсуждение, покажем, что эти две точки зрения определяют одно и то же множество. Мы распишем доказательство во всех подробностях, чтобы продемонстрировать, как различные его части соотносятся друг с другом.

Утверждение 3.2.6 $\mathcal{T} = S$.

Доказательство: \mathcal{T} определяется как наименьшее множество, удовлетворяющее некоторым условиям. Таким образом, достаточно показать, что (а) эти условия выполняются на S ; и (б) любое множество, удовлетворяющее условиям, содержит S как подмножество (т. е., что S — наименьшее множество, удовлетворяющее условиям).

В части (а) нам требуется проверить, что все три условия из определения 3.2.1 выполняются на S . Во-первых, поскольку $S_1 = \{\text{true}, \text{false}, 0\}$, ясно, что константы содержатся в S . Во-вторых, если $t_1 \in S$, то, поскольку $S = \bigcup_i S_i$, должно иметься некоторое i , такое, что $t_1 \in S_i$. Но тогда, по определению S_{i+1} , мы имеем $\text{succ } t_1 \in S_{i+1}$, а следовательно, $\text{succ } t_1 \in S$; аналогично, $\text{pred } t_1 \in S$ и $\text{iszero } t_1 \in S$. В-третьих, аналогичное рассуждение показывает, что если $t_1 \in S$, $t_2 \in S$, и $t_3 \in S$, то $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in S$.

Для части (б) предположим что некоторое множество S' удовлетворяет всем трем условиям из определения 3.2.1. При помощи полной индукции по i мы докажем, что все $S_i \subseteq S'$, откуда очевидно следует, что $S \subseteq S'$.

Допустим, что $S_j \subseteq S'$ для всех $j < i$; мы должны показать, что $S_i \subseteq S'$. Поскольку определение S_i состоит из двух пунктов (для $i = 0$ и $i > 0$), требуется рассмотреть оба случая. Если $i = 0$, то $S_i = \emptyset$; очевидно, что $\emptyset \subseteq S'$. В противном случае, $i = j + 1$ для некоторого j . Пусть t — некоторый элемент S_{j+1} . Поскольку S_{j+1} определяется как объединение трех меньших множеств, требуется рассмотреть три возможных случая. (1) Если t — константа, то $t \in S'$ по условию 1. (2) Если t имеет вид $\text{succ } t_1$, $\text{pred } t_1$ или $\text{iszero } t_1$, для некоторого $t_1 \in S_j$, то, по предположению индукции, $t_1 \in S'$, и тогда, по условию 2, $t \in S'$. (3) Если t имеет вид $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$ для некоторых $t_1, t_2, t_3 \in S_j$, то, опять же, согласно предположению индукции, t_1, t_2 и t_3 содержатся в S' , и, по условию 3, в нем содержится также и t .

Таким образом, мы показали, что все $S_i \subseteq S'$. Поскольку S определено как объединение всех S_i , мы имеем $S \subseteq S'$, что и завершает доказательство.

Стоит заметить, что доказательство использует *полную* индукцию по всем натуральным числам, а не более привычный образец «базовый случай / шаг индукции». Для каждого i мы предполагаем, что нужное условие выполняется для всех чисел строго меньше i , и доказываем, что и для i оно тоже выполняется. В сущности, *каждый* шаг здесь является шагом индукции; единственное, что выделяет случай $i = 0$ — это то, что множество чисел, меньших i , для которых мы можем использовать индуктивное предположение, оказывается пустым. То же соображение будет относиться к большинству индуктивных доказательств на всем протяжении этой книги — особенно к доказательствам по «структурной индукции».

3.3. Индукция на термах

Явная характеристика множества термов \mathcal{T} в Утверждении 3.2.6 позволяет нам сформулировать важный принцип изучения его элементов. Если $t \in \mathcal{T}$, то должно быть истинно одно из следующих утверждений: (1) t является константой, либо (2) t имеет вид $\text{succ } t_1$, $\text{pred } t_1$ или $\text{iszero } t_1$, причем t_1 меньше, чем t , либо, наконец (3) t имеет вид $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$, причем t_1 , t_2 и t_3 меньше, чем t . Это наблюдение можно использовать двумя способами: во-первых, мы можем строить *индуктивные определения* (inductive definitions) функций, действующих на множестве термов, а во-вторых, мы можем давать *индуктивные доказательства* (inductive proofs) свойств термов. Вот, например, индуктивное определение функции, которая ставит в соответствие каждому терму множество констант, в нем использованных.

Определение 3.3.1 Множество констант, встречающихся в терме t (записывается $\text{Consts}(t)$), определяется так:

$$\begin{aligned} \text{Consts}(\text{true}) &= \{\text{true}\} \\ \text{Consts}(\text{false}) &= \{\text{false}\} \\ \text{Consts}(0) &= \{0\} \\ \text{Consts}(\text{succ } t_1) &= \text{Consts}(t_1) \\ \text{Consts}(\text{pred } t_1) &= \text{Consts}(t_1) \\ \text{Consts}(\text{iszero } t_1) &= \text{Consts}(t_1) \\ \text{Consts}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{Consts}(t_1) \cup \text{Consts}(t_2) \cup \text{Consts}(t_3) \end{aligned}$$

Еще одна характеристика терма, которую можно вычислить при помощи индуктивного определения — его размер.

Определение 3.3.2 Размер (size) терма t (записывается $\text{size}(t)$) определяется так:

$$\begin{aligned} \text{size}(\text{true}) &= 1 \\ \text{size}(\text{false}) &= 1 \\ \text{size}(0) &= 1 \\ \text{size}(\text{succ } t_1) &= \text{size}(t_1) + 1 \\ \text{size}(\text{pred } t_1) &= \text{size}(t_1) + 1 \\ \text{size}(\text{iszero } t_1) &= \text{size}(t_1) + 1 \\ \text{size}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) + 1 \end{aligned}$$

Таким образом, размер t — это число вершин в его абстрактном синтаксическом дереве. Аналогично, глубина (depth) терма t (записывается $\text{depth}(t)$) определяется так:

$$\begin{aligned} \text{depth}(\text{true}) &= 1 \\ \text{depth}(\text{false}) &= 1 \\ \text{depth}(0) &= 1 \\ \text{depth}(\text{succ } t_1) &= \text{depth}(t_1) + 1 \\ \text{depth}(\text{pred } t_1) &= \text{depth}(t_1) + 1 \\ \text{depth}(\text{iszero } t_1) &= \text{depth}(t_1) + 1 \\ \text{depth}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \max(\text{depth}(t_1), \text{depth}(t_2), \text{depth}(t_3)) + 1 \end{aligned}$$

Другое, эквивалентное, определение, таково: $\text{depth}(t)$ — это наименьшее i , такое что $t \in S_i$ согласно определению 3.2.1.

Вот пример индуктивного доказательства простого соотношения между числом констант в терме и его размером. (Само свойство, разумеется, совершенно очевидно. Интерес представляет форма индуктивного рассуждения, которую мы еще неоднократно встретим.)

Лемма 3.3.3 Число различных констант в терме t не больше его размера (т. е., $|\text{Consts}(t)| \leq \text{size}(t)$).

Доказательство: индукция по глубине терма t . В предположении, что свойство выполняется для всех термов, меньших t , следует доказать его по отношению к самому t . Требуется рассмотреть три варианта:

Вариант: t — константа.

Свойство выполняется непосредственно: $|\text{Consts}(t)| = |\{t\}| = 1 = \text{size}(t)$.

Вариант: $t = \text{succ } t_1, \text{pred } t_1$ или $\text{iszero } t_1$

Согласно индуктивному предположению, $|\text{Consts}(t_1)| \leq \text{size}(t_1)$. Рассуждаем так: $|\text{Consts}(t)| = |\text{Consts}(t_1)| \leq \text{size}(t_1) < \text{size}(t)$.

Вариант: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

Согласно индуктивному предположению, $|\text{Consts}(t_1)| \leq \text{size}(t_1)$, $|\text{Consts}(t_2)| \leq \text{size}(t_2)$, $|\text{Consts}(t_3)| \leq \text{size}(t_3)$. Рассуждаем так:

$$\begin{aligned} |\text{Consts}(t)| &= |\text{Consts}(t_1) \cup \text{Consts}(t_2) \cup \text{Consts}(t_3)| \\ &\leq |\text{Consts}(t_1)| + |\text{Consts}(t_2)| + |\text{Consts}(t_3)| \\ &\leq \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) \\ &< \text{size}(t) \end{aligned}$$

Принцип, лежащий в основе этого доказательства, можно переформулировать ещё яснее, в качестве общего принципа рассуждения. Добавим для полноты еще два подобных принципа, часто используемых в доказательствах утверждений о термах.

Теорема 3.3.4 [ПРИНЦИПЫ ИНДУКЦИИ ПО ТЕРМАМ] Пусть P — предикат, определенный на множестве термов.

Индукция по глубине:

Если для каждого терма s ,

предполагая, что $P(r)$ для всех r , таких, что $\text{depth}(r) < \text{depth}(s)$,
можно доказать, что $P(s)$,

то $P(s)$ выполняется для всех s .

Индукция по размеру:

Если для каждого терма s ,

предполагая, что $P(r)$ для всех r , таких, что $\text{size}(r) < \text{size}(s)$,
можно доказать, что $P(s)$,
то $P(s)$ выполняется для всех s .

Структурная индукция:

Если для каждого терма s ,
предполагая, что $P(r)$ для всех непосредственных подтермов s , можно доказать, что $P(s)$,
то $P(s)$ выполняется для всех s .

Доказательство: УПРАЖНЕНИЕ ($\star\star$).

Индукция по глубине или размеру термов аналогична полной индукции на натуральных числах (2.4.2). Обыкновенная структурная индукция соответствует принципу обыкновенной индукции на натуральных числах (2.4.1), в которой шаг индукции требует, чтобы $P(n + 1)$ выводилось исключительно из предположения $P(n)$.

Подобно различным видам индукции на натуральных числах, выбор конкретного принципа индукции диктуется тем, какой из них приводит к более простой структуре конкретного доказательства — с формальной точки зрения, они выводятся друг из друга. В простых доказательствах обычно неважно, проводим ли мы индукцию по глубине, размеру или структуре. С точки зрения стиля, как правило, по возможности предпочитают структурную индукцию, поскольку в таком случае работа ведется прямо с термами, без промежуточного обращения к числам.

Большинство доказательств по индукции на термах имеет одну и ту же структуру. На каждом шаге индукции дается терм t , для которого нужно продемонстрировать некоторое свойство P , в предположении, что P выполняется на всех его подтермах (или на всех термах меньшего размера). Это делается путем рассмотрения всех возможных форм, которые может иметь t (`true`, `false`, 0, условное выражение и т. п.), доказывая в каждом случае, что P должно выполняться на всех термах этого вида. Поскольку от одного доказательства к другому меняются лишь детали рассуждений в конкретных случаях, принято опускать повторяющиеся части доказательств, и записывать их таким образом:

Доказательство: Индукция по t :

Вариант: $t = \text{true}$
... доказываем, что $P(\text{true})\dots$

Вариант: $t = \text{false}$
... доказываем, что $P(\text{false})\dots$

Вариант: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
... доказываем, что $P(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)$, используя $P(t_1)$,
 $P(t_2)$ и $P(t_3)\dots$

(И так же для остальных синтаксических форм.)

Во многих индуктивных доказательствах (включая 3.3.3) даже такой уровень подробности излишен: в базовых случаях (для термов t , не имеющих подтермов) $P(t)$ очевидно, а в индуктивных случаях $P(t)$ выводится путем применения индуктивного предположения к подтермам t и сочетания результатов некоторым очевидным способом. Читателю оказывается проще воссоздать доказательство на ходу (перебирая правила грамматики и держа в уме индуктивное предположение), чем прочитать явно записанные шаги. В таких ситуациях в качестве доказательства вполне приемлема фраза «индукция по t ».

3.4. Семантические стили

Сформулировав с математической точностью синтаксис нашего языка, мы должны теперь дать столь же строгое определение тому, как вычисляются термы — т. е., *семантике* (semantics) нашего языка. Существует три основных подхода к формализации семантики:

1. *Операционная семантика* (operational semantics) специфицирует поведение языка программирования, определяя для него простую *абстрактную машину* (abstract machine). Машина эта «абстрактна» в том смысле, что в качестве машинного кода она использует термы языка, а не набор команд какого-то низкоуровневого микропроцессора. Для простых языков *состояние* (state) машины — это просто терм, а поведение ее определяется *функцией перехода* (transition function), которая для каждого состояния либо указывает следующее состояние, произведя шаг упрощения старого терма, либо объявляет машину остановившейся. *Смыслом* (meaning) терма t объявляется конечное состояние, которого машина достигает, будучи запущена с начальным состоянием t .²

Иногда полезно дать для одного и того же языка несколько различных операционных семантик — а) более абстрактных, машинные состояния которых похожи на термы, записываемые программистом, и б) более близких к структурам, которыми манипулирует настоящий компилятор или интерпретатор языка. Доказательство того, что результаты работы этих различных машин при выполнении одной и той же программы в некотором смысле соответствуют друг другу, равносильно доказательству корректности реализации языка.

2. *Денотационная семантика* (denotational semantics) рассматривает смысл с более абстрактной точки зрения: смыслом терма считается не последовательность машинных состояний, а некоторый математический

²Строго говоря, то, что мы здесь описываем, — это так называемая *операционная семантика с малым шагом* (small-step operational semantics), известная также как *структурная операционная семантика* (structural operational semantics) (Plotkin, 1981). В упражнении 3.5.17 вводится альтернативный стиль *с большим шагом* (big-step), называемый также *естественной семантикой* (natural semantics) (Kahn, 1987), где единственный переход абстрактной машины сразу вычисляет окончательное значение терма.

объект, например, число или функция. Построение денотационной семантики для языка состоит в нахождении некоторого набора *семантических доменов* (semantic domains), а также определении *функции интерпретации* (interpretation function), которая ставит элементы этих доменов в соответствие термам. Поиск подходящих семантических доменов для моделирования различных языковых конструкций привел к возникновению сложной и изящной области исследований, известной как *теория доменов* (domain theory).

Одно из важных преимуществ денотационной семантики состоит в том, что она абстрагируется от мелких деталей выполнения программы и концентрирует внимание на основных понятиях языка. Кроме того, свойства выбранного набора семантических доменов могут использоваться для выявления важных законов поведения программ — например, законов, утверждающих, что две программы ведут себя одинаково, или что поведение программы соответствует некоторой спецификации. Наконец, из свойств выбранного набора семантических доменов часто непосредственно ясно, что некоторые (желательные или нежелательные) вещи в данном языке невозможны.

3. *Аксиоматическая семантика* (axiomatic semantics) предполагает более прямой подход к этим законам: вместо того, чтобы сначала определить поведение программ (с помощью операционной или денотационной семантики), а затем выводить из этого определения законы, аксиоматические методы используют *сами* законы в качестве определения языка. Смысл терма — это то, что о нем можно доказать.

Красота аксиоматических методов в том, что они концентрируют внимание на процессе рассуждений о программах. Именно эта традиция мышления обогатила информатику такими мощными инструментами, как *инварианты* (invariants).

В 60-е и 70-е годы считалось, что операционная семантика уступает двум другим стилям; что она полезна для быстрого и грубого определения характеристик языка, но в целом менее изящна и слаба с математической точки зрения. Однако в 80-е годы более абстрактные методы стали сталкиваться со все более неприятными техническими сложностями,³ и простота и гибкость операционной семантики стали по сравнению с ними выглядеть все более привлекательными — особенно в свете новых достижений, начиная со «Структурной операционной семантики» (Structural Operational Semantics) Плоткина (1981), «Естественной семантики» (Natural Semantics) Кана (1987) и работ Милнера по «Исчислению взаимодействующих систем» (Calculus of Communicating Systems, CCS; (1980; 1989; 1999)). В этих работах были введены в обращение новые изящные формализмы, и было показано, как перенести на операционную почву многие мощные методы, изначально разработанные в рамках денотационной семантики. Операционная семантика стала быстро развиваться и часто выбирается в качестве средства для определения языков программирования и изучения их свойств. Именно она используется в этой книге.

³ Для денотационной семантики камнем преткновения оказались недетерминистские вычисления и параллелизм; для аксиоматической семантики — процедуры.

В (бестиповое)

Синтаксис		Вычисление
$t ::=$	термы:	$t \rightarrow t'$
true	константа «истина»	$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2$
false	константа «ложь»	$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3$
if t then t_1 else t_2	условное	(E-IFTRUE)
	выражение	(E-IFFALSE)
$v ::=$	значения:	
true	константа «истина»	$t_1 \rightarrow t'_1$
false	константа «ложь»	$\frac{\text{if } t_1 \text{ then } t_2 \text{ else } t_3}{\rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$
		(E-IF)

Рис. 3.1. Булевские выражения (В)

3.5. Вычисление

Забудем на время о натуральных числах и рассмотрим операционную семантику только булевых выражений. Определение приведено на рис. 3.1. Рассмотрим его в деталях.

В левом столбце рис. 3.1 приведена грамматика, в которой определяются два набора выражений. Во-первых, повторяется (для удобства) синтаксис термов. Во-вторых, определяется подмножество термов — множество *значений* (values), которые являются возможными результатами вычисления. В нашем случае значениями являются только константы **true** и **false**. На протяжении всей книги для значений используется метапеременная v .

В правом столбце определяется *отношение вычисления* (evaluation relation)⁴ на термах. Оно записывается в виде $t \rightarrow t'$ и читается « t за один шаг вычисляется в t' ». Интуитивно ясно, что если в какой-то момент абстрактная машина находится в состоянии t , то она может произвести шаг вычисления и перейти в состояние t' . Это отношение определяется тремя правилами вывода (или, если угодно, двумя аксиомами и одним правилом: первые два правила не имеют предпосылок).

Первое правило, E-IFTURE, говорит, что при вычислении условного выражения, в котором условием служит константа **true**, машина может выкинуть это условное выражение, оставив истинную ветвь t_2 в качестве состояния (т. е., следующего подлежащего вычислению терма). Аналогично, правило E-IFFALSE говорит, что условное выражение с константой **false** в качестве

⁴Некоторые специалисты предпочитают называть это отношение *редукцией* (reduction), а термин *вычисление* (evaluation) сохранять для варианта «с большим шагом», который описан в упражнении 3.5.17 и напрямую сопоставляет термы и их окончательные значения.

условия переходит за один шаг в свою ложную ветвь t_3 . Префикс E- в названиях этих правил напоминает, что они являются частью отношения вычисления; у правил, определяющих другие отношения, будут в названиях другие префиксы.

Третье правило вычисления, E-IF, более интересно. Оно говорит, что если условие t_1 переходит за один шаг в t'_1 , то условное выражение `if t1 then t2 else t3` целиком переходит за шаг в `if t'_1 then t2 else t3`. Рассуждая в терминах абстрактных машин, машина, находящаяся в состоянии `if t1 then t2 else t3`, может за один шаг перейти в состояние `if t'_1 then t2 else t3`, если *другая* машина из состояния t_1 , может за шаг перейти в t'_1 .

То, о чём эти правила *не говорят*, не менее важно, чем то, что они говорят. Константы `true` и `false` ни во что не вычисляются, поскольку они не присутствуют в левой части никакого из правил. Более того, не существует правила, которое бы позволило вычислять истинную или ложную ветвь if-выражения, пока не вычислено условие: например, терм

```
if true then (if false then false else false) else true
```

не переходит в `if true then false else true`. Единственная разрешенная возможность — вычислить сначала внешнее условное выражение, используя E-IFTRUE. Такое взаимодействие между правилами определяет конкретную *стратегию вычисления* (evaluation strategy) для условных выражений, соответствующую привычному порядку вычислений в большинстве языков программирования: чтобы вычислить условное выражение, нужно вычислить его условие; если условие само является условным выражением, требуется вычислить *его* условие; и так далее. Правила E-IFTRUE и E-IFFALSE говорят нам, что нужно сделать, когда мы достигли конца этого процесса и обнаружили условное выражение, условие которого уже полностью вычислено. В некотором смысле, E-IFTRUE и E-IFFALSE проделывают работу по вычислению, в то время как E-IF помогает определить, где эта работа должна выполняться. Иногда разный характер этих правил подчеркивают, называя E-IFTRUE и E-IFFALSE *рабочими правилами* (computation rules), в то время как E-IF является *правилом соответствия* (congruence rule).

Для того, чтобы выразить наши интуитивные понятия более точно, можно формально определить отношение вычисления так:

Определение 3.5.1 Экземпляр (*instance*) правила вывода получается при замене каждой метапеременной одним и тем же термом в заключении правила и во всех его предпосылках (если они есть).

Например,

```
if true then true else (if false then false else false) → true
```

является экземпляром правила E-IFTRUE, в котором оба вхождения t_2 заменяются на `true`, а t_3 заменяется на `if false then false else false`.

Определение 3.5.2 Правило выполняется (*is satisfied*) на отношении, если для каждого экземпляра правила его заключение является элементом отношения, либо одна из его предпосылок не является таковой.

Определение 3.5.3 Одношаговое отношение вычисления (*one-step evaluation relation*) → есть наименьшее бинарное отношение на термах, на котором выполняются все три правила из рис. 3.1. Если пара (t, t') является элементом отношения вычисления, то мы говорим, что утверждение (или суждение) о вычислении $t \rightarrow t'$ выводимо (*the evaluation statement (judgement) is derivable*).

Значение слова «наименьшее» в этом определении таково: утверждение $t \rightarrow t'$ выводимо тогда и только тогда, когда оно обеспечено правилами: либо это экземпляр одной из аксиом E-IFTRUE или E-IFFALSE, либо это заключение экземпляра правила E-IF с выводимой предпосылкой. Выводимость утверждения можно обосновать через *дерево вывода* (derivation tree), в котором листьями служат экземпляры правил E-IFTRUE и E-IFFALSE, а внутренними вершинами — экземпляры правила E-IF. Например, если мы введем сокращения

$$\begin{aligned}s &\stackrel{\text{def}}{=} \text{if true then false else false} \\ t &\stackrel{\text{def}}{=} \text{if } s \text{ then true else true} \\ u &\stackrel{\text{def}}{=} \text{if false then true else true}\end{aligned}$$

(чтобы вывод помещался на странице), то выводимость утверждения

$$\text{if } t \text{ then false else false} \rightarrow \text{if } u \text{ then false else false}$$

можно обосновать при помощи дерева

$$\frac{\frac{\frac{s \rightarrow \text{false}}{\text{E-IFTRUE}}}{\frac{t \rightarrow u}{\text{E-IF}}} \text{E-IF}}{\text{if } t \text{ then false else false} \rightarrow \text{if } u \text{ then false else false}} \text{E-IF}$$

Может показаться странным, что мы называем такую структуру деревом, ведь в ней нет никакого ветвления. Действительно, деревья вывода, обосновывающие утверждения о вычислении, всегда будут иметь такую вырожденную форму: поскольку в правилах вычисления всегда не более одной предпосылки, то ветвлению в дереве вывода взяться неоткуда. Наша терминология станет более осмысленной, когда рассмотрение дойдет до других индуктивно определенных отношений, например, типизации — там некоторые правила могут иметь более одной предпосылки.

Тот факт, что утверждение о вычислении $t \rightarrow t'$ выводимо тогда и только тогда, когда существует дерево вывода с $t \rightarrow t'$ в корне, часто оказывается полезным при исследовании свойств отношения вычисления. В частности, это немедленно подсказывает метод доказательства, называемый *индукцией по выводам* (induction on derivations). Доказательство следующей теоремы может служить иллюстрацией этой методики.

Теорема 3.5.4 [ДЕТЕРМИНИРОВАННОСТЬ ОДНОШАГОВОГО ВЫЧИСЛЕНИЯ]

Если $t \rightarrow t' \wedge t \rightarrow t''$, то $t' = t''$.

Доказательство: *Индукция по выводу $t \rightarrow t'$. На каждом шаге индукции мы*

предполагаем теорему доказанной для всех меньших деревьев вывода, и анализируем правило вычисления, находящееся в корне дерева. (Заметим, что индукция здесь не идет по длине цепочки вычисления: мы смотрим только на один ее шаг. С тем же успехом можно было бы сказать, что индукция проводится по структуре t , поскольку структура «вывода шага вычисления» прямо повторяет структуру редуцируемого терма. Кроме того, мы могли бы точно так же провести индукцию по выводу $t \rightarrow t''$.)

Если последний шаг, использованный в выводе $t \rightarrow t'$, является экземпляром правила E-IFTRUE, то ясно, что t имеет вид $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$, причем $t_1 = \text{true}$. Но тогда понятно, что последний шаг в выводе $t \rightarrow t''$ не может быть E-IFFALSE, поскольку равенства $t_1 = \text{true}$ и $t_1 = \text{false}$ не могут выполняться одновременно. Более того, последнее правило во втором дереве вывода не может быть и E-IF, потому что предпосылка этого правила требует $t_1 \rightarrow t'_1$ для некоторого t'_1 , однако, как мы уже заметили, true не может ни во что перейти. Таким образом, последним правилом во втором дереве вывода может быть только E-IFTRUE, откуда немедленно следует, что $t' = t''$.

Точно так же, если последнее правило при выводе $t \rightarrow t'$ — экземпляр E-IFFALSE, таково же должно быть и последнее правило при выводе $t \rightarrow t''$, и результат, опять же, очевиден.

Наконец, если последним при выводе $t \rightarrow t'$ применяется правило E-IF, то из его формы можно заключить, что t имеет вид $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$, причем для некоторого t'_1 выполняется $t_1 \rightarrow t'_1$. При помощи таких же рассуждений, что и выше, мы можем заключить, что последним правилом в выводе $t \rightarrow t''$ может быть только E-IF, откуда ясно, что t имеет вид $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$ (что нам уже и так известно), и что для некоторого t''_1 выполняется $t_1 \rightarrow t''_1$. Но тогда мы можем применить предположение индукции (поскольку дерева вывода $t_1 \rightarrow t'_1$ и $t_1 \rightarrow t''_1$ являются поддеревьями выводов $t \rightarrow t'$ и $t \rightarrow t''$ соответственно) и заключить $t'_1 = t''_1$. Отсюда видно, что $t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 = \text{if } t''_1 \text{ then } t_2 \text{ else } t_3 = t''$, как нам и требуется.

Упражнение 3.5.5 [★] Сформулируйте в стиле теоремы 3.3.4 принцип индукции, используемый в приведенном выше доказательстве.

Наше одношаговое отношение вычисления показывает, как абстрактная машина переходит от одного состояния к другому при вычислении данного терма. Однако для нас, как для программистов, не меньший интерес представляют окончательные результаты вычисления — т. е. состояния, в которых машина не может сделать никаких дальнейших шагов.

Определение 3.5.6 Терм t находится в нормальной форме (*normal form*), если к нему не применимо никакое правило вычисления — т. е., если не существует такого t' , что $t \rightarrow t'$. (Для краткости мы иногда будем говорить « t является нормальной формой» вместо « t является термом в нормальной форме».)

Мы уже заметили, что в нашей текущей системе `true` и `false` — нормальные формы (поскольку во всех правилах вычисления левые части являются

условными выражениями, очевидно, невозможно построить экземпляр правила, где в качестве левой части выступали бы `true` или `false`). Можно обобщить это наблюдение в виде свойства всех значений:

Теорема 3.5.7 *Всякое значение находится в нормальной форме.*

Когда мы обогатим нашу систему арифметическими выражениями (а в последующих главах и другими конструкциями), мы всегда будем следить, чтобы теорема 3.5.7 оставалась истинной: существование в нормальной форме входит в смысл понятия значения, и всякое определение языка, где это не так, безнадежно испорчено.

В нашей теперешней системе верно и обратное утверждение: всякая нормальная форма есть значение. В общем случае это будет не так; в частности, как мы увидим далее в этой главе при рассмотрении арифметических выражений, нормальные формы, не являющиеся значениями, будут играть важную роль при анализе ошибок времени выполнения (run-time errors).

Теорема 3.5.8 *Если t находится в нормальной форме, то t — значение.*

Доказательство: Предположим, что t не является значением. При помощи структурной индукции на t легко показать, что t — не нормальная форма.

Если t — не значение, оно должно иметь вид `if` t_1 `then` t_2 `else` t_3 для некоторых t_1, t_2, t_3 . Рассмотрим возможные формы t_1 .

Если $t_1 = \text{true}$, то, очевидно, t не является нормальной формой, поскольку он подпадает под левую часть правила E-IFTRUE. Аналогично в случае $t_1 = \text{false}$.

Если t_1 не равно ни `true`, ни `false`, то оно не является значением. В таком случае применимо предположение индукции, утверждающее, что t_1 — не нормальная форма, а именно, что существует некий терм t'_1 , такой, что $t_1 \rightarrow t'_1$. Но тогда мы можем применить правило E-IF, получая $t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$. Таким образом, t не является нормальной формой.

Иногда оказывается удобно рассматривать несколько шагов вычисления как один большой переход между состояниями. Для этого мы определяем отношение многошагового вычисления, которое сопоставляет данному терму все термы, получаемые из него за ноль или более шагов.

Определение 3.5.9 Отношение многошагового вычисления (*multi-step evaluation relation*) \rightarrow^* — это рефлексивно-транзитивное замыкание отношения (одношагового) вычисления. То есть, это наименьшее отношение, такое, что (1) если $t \rightarrow t'$, то $t \rightarrow^* t'$, (2) для всех t выполняется $t \rightarrow^* t$, и (3) если $t \rightarrow^* t'$ и $t' \rightarrow^* t''$, то $t \rightarrow^* t''$.

Упражнение 3.5.10 [★] Переформулируйте определение 3.5.9 в виде набора правил вывода.

Явный способ записи многошагового вычисления облегчает формулировку, например, таких утверждений:

Теорема 3.5.11 [Единственность нормальных форм] *Если $t \rightarrow^* u$ и $t \rightarrow^* u'$, причем u и u' — нормальные формы, то $u = u'$.*

Доказательство: Теорема следует из детерминированности одношагового вычисления (3.5.4).

Последнее свойство вычислений, которое мы рассмотрим, прежде чем обратиться к арифметическим выражениям, таково: *каждый* терм можно вычислить и получить значение. Понятно, что это еще одна характеристика, которой могут не обладать более богатые языки, в которых, например, присутствуют рекурсивные определения функций. Даже в случаях, для которых свойство верно, доказательство его обычно гораздо сложнее, чем нижеприведенное доказательство. В главе 12 мы вернемся к этому вопросу и покажем, как система типов может служить основой доказательства гарантии завершения для некоторых языков.

Большинство доказательств гарантии завершения в информатике имеют одну и ту же базовую структуру.⁵ Сначала выберем некоторое вполне упорядоченное множество S и функцию f , переводящую «машинные состояния» (в нашем случае — термы) в элементы S . Затем покажем, что всякий раз, когда машинное состояние t может перейти в другое состояние t' , выполняется неравенство $f(t') < f(t)$. Теперь можно заметить, что бесконечная цепочка шагов вычисления, которая начинается с t , может быть переведена в бесконечную убывающую последовательность элементов S . Поскольку S вполне упорядочено, такая бесконечная убывающая цепочка существовать не может, а, соответственно, не может быть и бесконечной последовательности шагов вычисления. Функцию f часто называют *мерой завершения* (termination measure) отношения вычисления.

Теорема 3.5.12 [ЗАВЕРШЕНИЕ ВЫЧИСЛЕНИЙ] *Для каждого терма t существует нормальная форма t' такая, что $t \rightarrow^* t'$.*

Доказательство: Достаточно заметить, что *каждый* шаг вычисления уменьшает размер терма, и что размер может служить мерой завершения, поскольку обычный порядок на натуральных числах является полным.

Упражнение 3.5.13 [РЕКОМЕНДУЕТСЯ, ★★]

1. Предположим, что мы добавили к правилам на рис. 3.1 еще одно:

$$\text{if } \text{true} \text{ then } t_2 \text{ else } t_3 \rightarrow t_3 \text{ (E-FUNNY1)}$$

Какие из теорем 3.5.4, 3.5.7, 3.5.8, 3.5.11 и 3.5.12 остаются истинными?

2. Теперь допустим, что мы добавляем такое правило:

$$\frac{t_2 \rightarrow t'_2}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1 \text{ then } t'_2 \text{ else } t_3} \text{ (E-FUNNY2)}$$

Какие теоремы сохраняются в этом случае? Требуется ли модифицировать какие-либо доказательства?

⁵ В главе 12 мы увидим доказательство гарантии завершения с несколько более сложной структурой.

В N (бестиповое)

Расширяет **В** (3.1)

Новые синтаксисические формы

t	$::= \dots$	термы:
	0	константа ноль
	succ t	следующее число
	pred t	предыдущее число
	iszzero t	проверка на ноль
v	$::= \dots$	значения:
	nv	числовое значение
	nv	числовые значения:
	0	нулевое значение
	succ nv	значение-последователь

Новые правила вычисления

$$t \rightarrow t'$$

$$\frac{t_1 \rightarrow t'_1}{\text{succ } t_1 \rightarrow \text{succ } t'_1} \quad (\text{E-SUCC})$$

$$\text{pred } 0 \rightarrow 0 \quad (\text{E-PREDZERO})$$

$$\frac{\text{pred } (\text{succ } nv_1) \rightarrow nv_1}{\text{pred } t_1 \rightarrow \text{pred } t'_1} \quad (\text{E-PREDSUCC})$$

$$\frac{t_1 \rightarrow t'_1}{\text{pred } t_1 \rightarrow \text{pred } t'_1} \quad (\text{E-PRED})$$

$$\text{iszzero } 0 \rightarrow \text{true} \quad (\text{E-ISZEROZERO})$$

$$\frac{\text{iszzero } (\text{succ } nv_1) \rightarrow \text{false}}{\text{iszzero } t_1 \rightarrow \text{iszzero } t'_1} \quad (\text{E-ISZEROSUCC})$$

$$\frac{t_1 \rightarrow t'_1}{\text{iszzero } t_1 \rightarrow \text{iszzero } t'_1} \quad (\text{E-IsZERO})$$

Рис. 3.2. Арифметические выражения (NB)

Следующей нашей задачей будет расширение понятия вычисления на арифметические выражения. На рис. 3.2 продемонстрированы новые компоненты определения. (Пометка в правом верхнем углу рис. 3.2 напоминает, что определение расширяет 3.1, а не рассматривается независимо.)

Как и раньше, определение термов просто повторяет синтаксис, который мы уже видели в §3.1. Определение значений несколько интереснее, поскольку приходится вводить новую категорию *числовых значений* (syntactic values). Интуитивное представление таково: окончательным результатом вычисления арифметического выражения может быть число, причем под числом понимается либо 0, либо значение-последователь числа (но не последователь произвольного значения: `succ(true)` должно считаться ошибкой, а не значением).

Правила вычисления в правом столбце рис. 3.2 построены по той же схеме, что и на рис. 3.1. Четыре рабочих правила (E-PREDZERO, E-PREDSUCC, E-ISZEROZERO и E-ISZEROSUCC) показывают, как операторы `pred` и `iszzero` действуют на числовые значения, а три правила соответствия (E-SUCC, E-PRED и E-IsZERO) управляют вычислением «первого» подтерма составного терма.

Строго говоря, сейчас следовало бы повторить определение 3.5.3 («Одношаговое отношение вычисления \rightarrow есть наименьшее бинарное отношение на

термах, на котором выполняются все три правила из рис. 3.1 и 3.2...»). Чтобы не тратить место на такого рода служебные определения, обычно считают, что правила вывода сами по себе представляют определение, а формулировка «наименьшее отношение, содержащее все экземпляры...» подразумевается.

В этих правилах важную роль играет синтаксическая категория числовых значений (*nv*). Например, в E-PRED_{SUCC} требование, чтобы левая часть имела вид *pred* (*succ nv₁*), а не просто *pred* (*succ t₁*), приводит к тому, что это правило нельзя использовать для перевода *pred* (*succ (pred 0)*) в *pred 0*, поскольку для такого шага потребовалось бы заменить метапеременную *nv₁* на терм *pred 0*, который не является числовым значением. В результате единственный разрешенный шаг при вычислении терма *pred* (*succ (pred 0)*) имеет следующее дерево вывода:

$$\frac{\frac{\frac{\text{pred } 0 \rightarrow 0}{\text{succ } (\text{pred } 0) \rightarrow \text{succ } 0} \text{ E-SUCC}}{\text{pred } (\text{succ } (\text{pred } 0)) \rightarrow \text{pred } (\text{succ } 0)} \text{ E-PRED}}{\text{E-PREDZERO}}$$

Упражнение 3.5.14 [★★] Покажите, что теорема 3.5.4 верна и для отношения вычисления на арифметических выражениях: если $t \rightarrow t'$ и $t \rightarrow t''$, то $t' = t''$.

Формализация операционной семантики языка требует определения поведения *всех* термов, включая такие термы нашего языка, как *pred 0* и *succ false*. По правилам рис. 3.2, предшествующим числом 0 считается 0. С другой стороны, последователь *false* не вычисляется никак (другими словами, этот терм является нормальной формой). Такие термы мы называем *туниками*.

Определение 3.5.15 Терм называется *туниковым* (*stuck*), если он находится в нормальной форме, но не является значением.

Понятие «туникового терма» представляет в нашей простой машине *ошибки времени выполнения* (run-time errors). С интуитивной точки зрения, оно описывает ситуации, когда операционная семантика не знает, что делать дальше, поскольку программа оказалась в «бессмысленном» состоянии. В более конкретной реализации языка такие состояния могут соответствовать различного рода ошибкам: обращениям по несуществующему адресу, попыткам выполнить запрещенную машинную команду и т. п. Мы объединяем все эти виды неправильного поведения в единую категорию «туникового состояния».

Упражнение 3.5.16 [РЕКОМЕНДУЕТСЯ, ★★★]

Другой способ формализации бессмысленных состояний абстрактной машины состоит в том, чтобы ввести новый терм *stop* («неправильное значение») и дополнить операционную семантику правилами, которые бы явным

образом порождали *wrong* во всех ситуациях, которые в нынешней семантике приводят к тупику. А именно, мы вводим две новые синтаксические категории

<i>badnat</i>	::=	нечисловые нормальные формы
	<i>wrong</i>	ошибка времени выполнения
	<i>true</i>	константа «истина»
	<i>false</i>	константа «ложь»
<i>badbool</i>	::=	небулевские нормальные формы
	<i>wrong</i>	ошибка времени выполнения
	<i>nv</i>	числовое значение

и дополняем отношение вычисления следующими правилами:

<i>if badbool then t₁ else t₂</i> →	<i>wrong</i>	(E-IF-WRONG)
<i>succ badnat</i> →	<i>wrong</i>	(E-SUCC-WRONG)
<i>pred badnat</i> →	<i>wrong</i>	(E-PRED-WRONG)
<i>iszzero badnat</i> →	<i>wrong</i>	(E-IsZERO-WRONG)

Покажите, что эти два подхода к формализации ошибок времени выполнения согласуются между собой. Для этого нужно (1) найти способ точного выражения интуитивной идеи о том, что «два подхода согласуются», и (2) доказать ее. Как это часто бывает при изучении языков программирования, сложнее всего сформулировать точное утверждение, подлежащее доказательству — само доказательство после этого построить нетрудно.

Упражнение 3.5.17 [Рекомендуется, ★★★]

Обычно используется два различных стиля операционной семантики. В этой книге используется так называемая семантика с малым шагом (*small-step*): в определении отношения вычисления показано, как отдельные шаги вычисления используются для переписывания частей терма, фрагмент за фрагментом, пока в конце концов не получится значение. На базе этого отношения мы определяем многошаговое отношение вычисления, которое позволяет нам говорить о том, как термы (за много шагов) вычисляются и дают значения. Другой стиль — семантика с большим шагом (*big-step*) (или, иногда, естественная семантика, *natural semantics*), прямо определяет понятие «терм такой-то при вычислении дает такое-то значение», которое записывается как $t \Downarrow v$. Правила с большим шагом для нашего языка с булевскими и арифметическими выражениями выглядят так:

$$v \Downarrow v \quad (\text{B-VALUE})$$

$$\frac{t_1 \Downarrow \text{true} \quad t_2 \Downarrow v_2}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_2} \quad (\text{B-IFTTRUE})$$

$$\frac{t_1 \Downarrow \text{false} \quad t_3 \Downarrow v_3}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_3} \quad (\text{B-IFFALSE})$$

$$\frac{t_1 \Downarrow nv_1}{succ\ t_1 \Downarrow succ\ nv_1} \quad (\text{B-SUCC})$$

$$\frac{t_1 \Downarrow 0}{pred\ t_1 \Downarrow 0} \quad (\text{B-PREDZERO})$$

$$\frac{t_1 \Downarrow succ\ nv_1}{pred\ t_1 \Downarrow nv_1} \quad (\text{B-PREDSUCC})$$

$$\frac{t_1 \Downarrow 0}{iszero\ t_1 \Downarrow true} \quad (\text{B-ISZEROTRUE})$$

$$\frac{t_1 \Downarrow succ\ nv_1}{iszero\ t_1 \Downarrow false} \quad (\text{B-ISZEROFALSE})$$

Покажите, что семантика с малым шагом и с большим шагом для нашего языка дают один и тот же результат, т. е., $t \rightarrow^* v$ тогда и только тогда, когда $t \Downarrow v$.

Упражнение 3.5.18 [$\star\star \rightarrow$]

Предположим, что нам захотелось поменять стратегию вычисления для нашего языка, так, чтобы ветви *then* и *else* в условном выражении вычислялись (в указанном порядке) до того, как вычислится само условие. Покажите, как нужно изменить правила вычисления, чтобы добиться такого поведения.

3.6. Дополнительные замечания

Понятия абстрактного и конкретного синтаксиса, синтаксического анализа и т. п. объясняются во многих учебниках по компиляторам. Индуктивные определения, системы правил вывода и доказательства по индукции рассмотрены более подробно в книгах Винскеля (1993) и Хенnessy (Hennessy, 1990).

Стиль операционной семантики, которым мы здесь пользуемся, восходит к техническому отчету Плоткина (1981). Стиль с большим шагом (3.5.17) был разработан Каном (1987). Более подробно они описаны у Астезиано (Astesiano, 1991) и у Хеннесси (Hennessy, 1990).

Структурную индукцию ввел в информатику Берсталл (Burstall, 1969).

В.: Зачем доказывать свойства языков программирования?
Если в определениях нет ошибок, доказательства почти всегда получаются очень скучными.
О.: В определениях почти всегда есть ошибки.

Автор неизвестен

Глава 4

Реализация арифметических выражений на языке ML

С формальными определениями, такими как те, что мы видели в предыдущей главе, работать часто проще, если интуитивные понятия, которые за ними стоят, «подтверждены» конкретной реализацией. В этой главе мы описываем основные компоненты интерпретатора для нашего языка логических и арифметических выражений. (Те читатели, которые не собираются работать с описанными в этой книге реализациями программ проверки типов, могут пропустить эту главу и все последующие, в заглавии которых упоминается «реализация на языке ML».)

Код, представленный здесь (и во всех разделах этой книги, посвященных реализации), написан на популярном языке из семейства ML (Gordon, Milner, and Wadsworth, 1979), который называется *Objective Caml*, или, сокращенно, *OCaml* (Leroy, 2000; Cousineau and Mauny, 1998). Используется лишь небольшое подмножество языка OCaml; все программы легко можно переписать на любой другой язык. Основные требования к языку — автоматическое управление памятью (сборка мусора) и простота определения рекурсивных функций через сопоставление с образцом на структурных типах данных. Вполне подходят другие функциональные языки, например, Standard ML (Milner, Tofte, Harper, and MacQueen, 1997), Haskell (Hudak et al., 1992; Thompson, 1999) или Scheme (Kelsey, Clinger, and Rees, 1998; Dybvig, 1996) (с каким-либо расширением для сопоставления с образцом). Языки со сборкой мусора, но без сопоставления, например, Java (Arnold and Gosling, 1996) или чистая Scheme, для наших задач несколько тяжеловаты. Языки, в которых нет ни того, ни другого, как C (Kernighan and Ritchie, 1988), подходят еще меньше.¹

Код из этой главы хранится в веб-репозитории по адресу <http://www.cis.upenn.edu/~bcpierce/tapl> под названием `arith`. Там же можно найти инструкций по скачиванию и сборке интерпретаторов.

¹ Разумеется, языковые вкусы бывают разные, и хороший программист способен работать с любым инструментом; читатель может выбрать тот язык, который ему нравится. Однако следует иметь в виду, что при символьной обработке, характерной для программ проверки типов, ручное управление памятью особенно утомительно и чревато ошибками.

4.1. Синтаксис

Сначала необходимо определить тип OCaml-значений, представляющий термы. Эта задача легко решается с помощью механизма задания типов в OCaml: вот объявление, представляющее собой прямой перевод грамматики со с. 44.

```
type term =
  TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmZero of info
  | TmSucc of info * term
  | TmPred of info * term
  | TmIsZero of info * term
```

Конструкторы от `TmTrue` до `TmIsZero` соответствуют различным видам вершин в синтаксических деревьях типа `term`; тип, следующий за словом `of`, для каждого случая указывает количество поддеревьев, растущих из вершины данного вида.

Каждый узел абстрактного синтаксического дерева снабжен значением типа `info`, в котором записано место в файле и имя файла, в котором определена эта вершина. Эта информация создается процедурой синтаксического анализа при чтении входного файла и используется функциями распечатки, чтобы сообщить пользователю о местонахождении ошибки. С точки зрения основных алгоритмов вычисления, проверки типов и т. п., эту информацию можно было бы и вовсе не хранить; она включена в листинги, чтобы читатели, которым захочется самим поэкспериментировать с реализациями, видели код точно в таком виде, в каком он напечатан в книге.

При определении отношения вычисления необходимо проверять, является ли терм числовым значением:

```
let rec isnumericval t = match t with
  TmZero(_) → true
  | TmSucc(_,t1) → isnumericval t1
  | _ → false
```

Это типичный пример рекурсивного определения через сопоставление с образцом в OCaml: `isnumericval` определяется как функция, которая, будучи вызвана с аргументом `TmZero`, возвращает `true`; будучи применена к `TmSucc` с поддеревом `t1`, вызывает себя рекурсивно с аргументом `t1`, чтобы проверить, является ли он числовым значением; а будучи вызвана с любым другим аргументом, возвращает `false`. Знаки подчеркивания в некоторых образцах — метки «неважно», они сопоставляются с любым термом, который стоит в указанном месте; в первых двух выражениях с их помощью игнорируются аннотации `info`, а в последнем выражении с помощью такой метки сопоставляется любой терм. Ключевое слово `rec` говорит компилятору, что определение функции рекурсивно — т. е., что идентификатор `isnumericval` в теле функции относится к самой функции, которую сейчас определяют, а не к какой-либо более ранней переменной с тем же именем.

Заметим, что ML-код в приведенном определении при верстке немного «украшен», как для простоты чтения, так и для сохранения стиля примеров из лямбда-исчисления. В частности, мы используем настоящий символ стрелки \rightarrow , а не двухсимвольную последовательность $->$.

Аналогично выглядит функция, проверяющая, является ли терм значением:

```
let rec isval t = match t with
  TmTrue(_) → true
  | TmFalse(_) → true
  | t when isnumericval t → true
  | _ → false
```

В третьем предложении встречается «условный образец»: он сопоставляется с любым термом t , но только если булевское выражение `isnumericval t` возвращает значение «истина».

4.2. Вычисление

Реализация отношения вычисления точно следует правилам одношагового вычисления по рис. 3.1 и 3.2. Как мы видели, эти правила определяют *частичную функцию* (partial function), которая, будучи применена к терму, не являющемуся значением, выдает следующий шаг вычисления этого терма. Если функцию попытаться применить к значению, она никакого результата не выдает. При переводе правил вычисления на OCaml нам нужно решить, как действовать в таком случае. Очевидный вариант — написать функцию одношагового вычисления `eval1` так, чтобы она вызывала исключение, если ни одно из правил невозможно применить к терму, полученному в качестве параметра. (Другой способ действий заключается в том, чтобы заставить вычислитель возвращать значение типа `term option`, которое бы показывало, было ли вычисление успешным, и если да, то каков его результат; этот вариант тоже работал бы, но потребовал бы больше служебного кода.) Определим сначала исключение, которое вызывается, если ни одно правило не применимо:

```
exception NoRuleApplies
```

Теперь можно написать саму функцию одношагового вычисления.

```
let rec eval1 t = match t with
  TmIf(_, TmTrue(_), t2, t3) →
    t2
  | TmIf(_, TmFalse(_), t2, t3) →
    t3
  | TmIf(fi, t1, t2, t3) →
    let t1' = eval1 t1 in
      TmIf(fi, t1', t2, t3)
  | TmSucc(fi, t1) →
    let t1' = eval1 t1 in
      TmSucc(fi, t1')
  | TmPred(_, TmZero(_)) →
    TmZero(dummyinfo)
```

```

| TmPred(_, TmSucc(_, nv1)) when (isnumericval nv1) →
  nv1
| TmPred(fi, t1) →
  let t1' = eval1 t1 in
  TmPred(fi, t1')
| TmIsZero(_, TmZero(_)) →
  TmTrue(dummyinfo)
| TmIsZero(_, TmSucc(_, nv1)) when (isnumericval nv1) →
  TmFalse(dummyinfo)
| TmIsZero(fi, t1) →
  let t1' = eval1 t1 in
  TmIsZero(fi, t1')
| _ →
  raise NoRuleApplies

```

Заметим, что в нескольких местах мы создаем новые термы, а не переделываем уже существующие. Так как этих новых термов нет в пользовательских исходных файлах, то аннотации `info` в них не имеют смысла. В таких термах мы используем константу `dummyinfo`. Для сопоставления с аннотациями в образцах всегда используется переменная `fi` (file information, «информация о файле»).

Еще одна существенная деталь в определении `eval1` — использование явных выражений `when` в образцах для передачи смысла имен метапеременных вроде `v` и `nv` в правилах, определяющих отношение вычисления на рис. 3.1 и 3.2. Например, в выражении для вычисления `TmPred(_, TmSucc(_, nv1))`, семантика образцов OCaml позволяет `nv1` сопоставляться с каким угодно термом, а мы этого не хотим; добавление `when (isnumericval nv1)` ограничивает правило так, чтобы оно срабатывало только если терм, соответствующий `nv1`, действительно является числовым значением. (Мы, в принципе, могли бы переписать исходные правила вывода в том же стиле, что и образцы ML, превратив неявные ограничения, подразумеваемые именами метапеременных, в явные условия применения правил:

$$\frac{t_1 \text{ — числовое значение}}{\text{pred } (\text{succ } t_1) \rightarrow t_1} \quad (\text{E-PREDSUCC})$$

При этом несколько пострадала бы компактность правил и их удобство для чтения.)

Наконец, функция `eval` берет терм и находит его нормальную форму, циклически применяя `eval1`. Если `eval1` возвращает новый терм `t'`, то мы вызываем `eval` рекурсивно, чтобы продолжить вычисление, начиная с `t'`. Когда, наконец, `eval1` достигнет состояния, где не применимо никакое правило, она вызывает исключение `NoRuleApplies`. При этом `eval` выходит из цикла и возвращает заключительный терм последовательности.²

²Мы записываем `eval` таким образом из соображений удобства для чтения, однако на самом деле помещать обработчик исключений `try` внутрь рекурсивного цикла в ML не рекомендуется.

Упражнение 4.2.1 [★★]:
Почему? Как было бы правильнее записать `eval`?

```
let rec eval t =
  try let t' = eval1 t
      in eval t'
  with NoRuleApplies → t
```

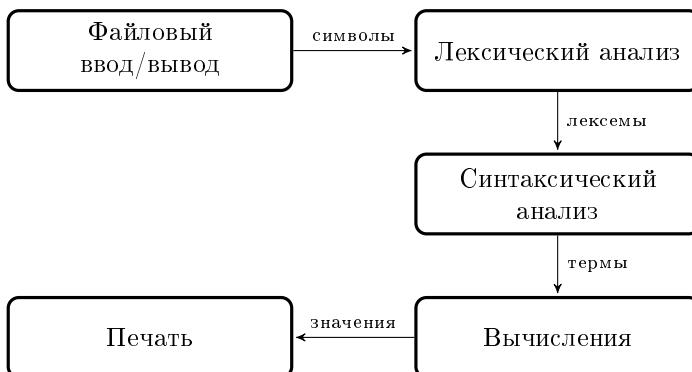
Разумеется, наш простой вычислитель записан так, чтобы упростить возможность его сравнения с математическим определением вычисления, а не так, чтобы находить нормальные формы как можно быстрее. Несколько более эффективный алгоритм можно получить на основе правил вычисления «с большим шагом», как предлагается в упражнении 4.2.2.

Упражнение 4.2.2 [РЕКОМЕНДУЕТСЯ, ★★★→]:

Измените определение функции eval в программе arith, используя стиль с большим шагом из упражнения 3.5.17.

4.3. Что осталось за кадром

Разумеется, в любом интерпретаторе и компиляторе — даже в очень простом, — много модулей помимо тех, которые мы обсудили выше. На самом деле вычисляемые термы изначально хранятся в файлах как последовательности символов. Их нужно считать из файловой системы, перевести в последовательность лексем при помощи лексического анализатора, потом с помощью синтаксического анализатора собрать из них абстрактные синтаксические деревья, и только после этого передать функциям вычисления, которые мы видели в этой главе. Более того, после вычисления нужно распечатать результаты.



Интересующиеся читатели могут исследовать имеющийся в сети код всего интерпретатора на OCaml.

Глава 5

Бестиповое лямбда-исчисление

В этой главе мы вспомним определение и некоторые базовые свойства *бестипового* (untyped), или *чистого* (pure), *лямбда-исчисления* (lambda-calculus). Бестиповое лямбда-исчисление служит вычислительной основой, «почвой», из которой появилось большинство систем типов, описываемых в оставшейся части книги.

В середине 60-х годов Питер Ландин отметил, что сложный язык программирования можно изучать, сформулировав его ядро в виде небольшого базового исчисления, которое выражает самые существенные механизмы языка, и дополнив его набором удобных *производных форм* (derived forms), поведение которых можно выразить путем перевода на язык базового исчисления (Landin 1964, 1965, 1966; см. также Tennent 1981). В качестве базового языка Ландин использовал *лямбда-исчисление* (lambda-calculus) — формальную систему, изобретенную в 1920-е годы Алонсо Чёрчем (Church, 1936, 1941), где все вычисление сводится к элементарным операциям — определению функции и ее применению. Под влиянием идей Ландина, а также новаторских работ Джона Маккарти по языку Lisp (McCarthy, Russell, Edwards, et al., 1959; McCarthy, 1981) лямбда-исчисление стало широко использоваться для спецификации конструкций языков программирования, в разработке и реализации языков, а также в исследовании систем типов. Важность этого исчисления состоит в том, что его можно одновременно рассматривать как простой язык программирования, *на котором* можно описывать вычисления, и как математический объект, *о котором* можно доказывать строгие утверждения.

Лямбда-исчисление — лишь одно из нескольких фундаментальных исчислений, используемых для подобных целей. *Пи-исчисление* (pi-calculus) Милнера, Парроу и Уокера (Milner, Parrow, and Walker, 1992; Milner, 1991)) завоевало популярность как базовый язык для определения семантики языков параллельных вычислений с обменом сообщениями, а *исчисление объектов*

Примеры из этой главы являются термами чистого бестипового лямбда-исчисления, λ (см. рис. 5.3), либо лямбда-исчисления с добавлением булевых значений и арифметических операций, λNB (3.2). Соответствующая реализация на OCaml называется **fulluntyped**.

(object calculus) Абади и Карделли (Abadi and Cardelli, 1996) выражает суть объектно-ориентированных языков. Большинство идей и методов, которые мы опишем и разработаем для лямбда-исчисления, можно без особого труда перенести и на эти другие исчисления. Один из примеров такого переноса представлен в главе 19.

Лямбда-исчисление можно расширить и обогатить несколькими способами. Во-первых, часто для удобства добавляют особый синтаксис для чисел, кортежей, записей и т. п., чье поведение, в принципе, можно смоделировать и в базовом языке. Интереснее добавить более сложные возможности, такие как изменяемые ссылочные ячейки или нелокальная обработка исключений. Эти свойства можно смоделировать на базовом языке только путем достаточно тяжеловесного перевода. Такие расширения, в конце концов, приводят к языкам вроде ML (Gordon, Milner, and Wadsworth, 1979; Milner, Tofte, and Harper, 1990; Weis, Aponte, Laville, Mauny, and Suárez, 1989; Milner, Tofte, Harper, and MacQueen, 1997), Haskell (Hudak et al., 1992) или Scheme (Sussman and Steele, 1975; Kelsey, Clinger, and Rees, 1998). Как мы увидим в последующих главах, расширения базового языка часто требуют расширения системы типов.

5.1. Основы

Процедурная (или функциональная) абстракция — ключевое свойство почти всех языков программирования. Вместо того, чтобы выписывать одно и то же вычисление раз за разом, мы пишем процедуру или функцию, которая проделывает это вычисление в общем виде, используя один или несколько именованных параметров, а затем при необходимости вызываем эту процедуру, каждый раз задавая значения параметров. К примеру, для программиста естественно взять длинное вычисление с повторяющимися частями, вроде

```
(5*4*3*2*1) + (7*6*5*4*3*2*1) - (3*2*1)
```

и переписать его в виде `factorial(5) + factorial(7) - factorial(3)`, где

```
factorial(n) = if n=0 then 1 else n * factorial(n-1)
```

Для каждого неотрицательного числа `n` подстановка аргумента `n` в функцию `factorial` дает в результате факториал `n`. Используя нотацию « $\lambda n. \dots$ » обозначающую «функцию, которая для каждого `n`, дает \dots », определение `factorial` можно переформулировать как

```
factorial =  $\lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n * \text{factorial}(n-1)$ 
```

Теперь `factorial(0)` означает «функция $\lambda n. \text{if } n=0 \text{ then } 1 \text{ else } \dots$, примененная к аргументу 0», то есть, «значение, которое получается, если аргумент `n` в теле функции ($\lambda n. \text{if } n=0 \text{ then } 1 \text{ else } \dots$) заменить на 0», то есть, «`if 0=0 then 1 else ...`», то есть, 1.

Лямбда-исчисление (lambda-calculus) (или λ -исчисление) воплощает такой способ определения и применения функций в наиболее чистой форме. В лямбда-исчислении все являются функциями: аргументы, которые функции принимают — тоже функции, и результат, возвращаемый функцией — опять-таки функция.

Синтаксис лямбда-исчисления состоит из трех видов термов.¹ Переменная x сама по себе есть терм; абстракция переменной x в терме t_1 , (записывается как $\lambda x. t_1$), — тоже терм; и, наконец, применение терма t_1 к терму t_2 (записывается $t_1 t_2$) — третий вид термов. Эти способы конструирования термов выражаются следующей грамматикой:

$t ::=$	<i>термы:</i>
x	<i>переменная</i>
$\lambda x. t$	<i>абстракция</i>
$t t$	<i>применение</i>

В следующих подразделах это определение изучается детально.

Абстрактный и конкретный синтаксис

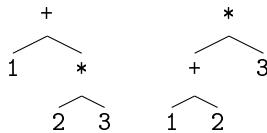
При обсуждении синтаксиса языков программирования полезно различать два уровня² структуры. *Конкретный синтаксис* (concrete syntax, или surface syntax) языка относится к строкам символов, которые непосредственно читают и пишут программисты. *Абстрактный синтаксис* (abstract syntax) — это намного более простое внутреннее представление программ в виде помеченных деревьев (они называются *абстрактными синтаксическими деревьями* (abstract syntax trees) или *АСД* (AST)). Представление в виде дерева делает структуру термов очевидной, и поэтому его удобно использовать для сложных преобразований, которые нужны как при строгом определении языков (и доказательстве их свойств), так и внутри компиляторов и интерпретаторов.

Преобразование из конкретного в абстрактный синтаксис происходит в два этапа. Сначала *лексический анализатор* (lexical analyzer) (или *лексер*, lexer) переводит последовательность символов, написанных программистом, в последовательность *лексем* (tokens) — идентификаторов, ключевых слов, комментариев, символов пунктуации, и т. п. Лексический анализатор убирает комментарии, обрабатывает пробелы, решает вопрос с заглавными и строчными буквами, а также распознает форматы числовых и символьных констант. После этого *грамматический анализатор* (parser) (parser) преобразует последовательность лексем в абстрактное синтаксическое дерево. При грамматическом разборе соглашения о *приоритете* (precedence) и *ассоциативности* (associativity) операторов помогают уменьшить количество скобок в программе, явно указывающих структуру составных выражений. Например, оператор $*$ имеет приоритет выше, чем оператор $+$, так что анализатор интерпретирует выражение без скобок $1+2*3$ как абстрактное синтаксическое дерево,

¹Выражение «*лямбда-терм* (lambda-term)» относится ко всем термам лямбда-исчисления. Лямбда-термы, которые начинаются с буквы λ , часто называют «*лямбда-абстракциями* (lambda-abstractions)».

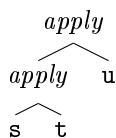
²Определения полноценных языков иногда используют еще большее число уровней. Например, вслед за Ландином, часто бывает полезно определять поведение некоторых конструкций языка в качестве производных форм, путем перевода их в комбинации других, более элементарных, конструкций. Ограниченный язык, состоящий только из этих базовых конструкций, часто называют *внутренним языком* (internal language), а полный язык, который включает в себя все производные формы, называется *внешним языком* (external language). Трансформация из внешнего языка во внутренний выполняется (по крайней мере, концептуально) отдельным проходом компилятора, вслед за синтаксическим анализом. Производные формы обсуждаются в разделе 11.3.

которое показано слева, а не справа:

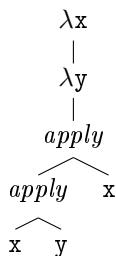


В этой книге мы обращаем основное внимание на абстрактный, а не конкретный синтаксис. Грамматики, вроде той, что мы привели для лямбда-термов, следует рассматривать как описания разрешенных видов деревьев, а не последовательностей лексем или символов. Разумеется, когда мы будем записывать термы в примерах, определениях, теоремах и доказательствах, нам придется выражать их с помощью конкретной, линейной записи, но мы всегда имеем в виду соответствующие абстрактные синтаксические деревья.

Чтобы избежать излишних скобок, для записи лямбда-термов в линейной форме мы следуем двум соглашениям. Во-первых, применение функции левоассоциативно — то есть, $s \ t$ и обозначает то же дерево, что $(s \ t)$ и:



Во-вторых, тела абстракций простираются направо как можно дальше, так что, например, $\lambda x. \lambda y. x \ y \ x$ означает то же самое, что и $\lambda x. (\lambda y. ((x \ y) \ x))$:



Переменные и метапеременные

Еще одна тонкость в приведенном определении синтаксиса касается использования метапеременных. Мы будем продолжать использовать метапеременную t (а также s и u , с нижними индексами и без них), обозначающую произвольный терм.³ Аналогично, x (а также y и z) замещает произвольную переменную. Заметим, что здесь x — это метапеременная, значениями которой являются другие переменные! К сожалению, число коротких имен

³Само собой, в этой главе t обозначает лямбда-терм, а не арифметическое выражение. На протяжении всей книги t будет обозначать терм того исчисления, которое обсуждается в данный момент. В начале каждой главы есть примечание, где указано, какая система рассматривается в этой главе.

ограничено, и нам потребуется иногда использовать x , y и т. д. для обозначения переменных объектного языка. В таких случаях, однако, из контекста всегда будет ясно, что имеется в виду. Например, в предложении «Терм $\lambda x. \lambda y. x y$ имеет вид $\lambda z. s$, где $z = x$, а $s = \lambda y. x y$ » имеем z и s — имена метапеременных, а x и y — имена переменных объектного языка.

Область видимости

Последнее, что нам требуется разъяснить в синтаксисе лямбда-исчисления, — *область видимости* (scope) переменных.

Переменная x называется *связанной* (bound), если она находится в теле t абстракции $\lambda x. t$. (Точнее, оно связано *этой* абстракцией. Мы можем также сказать, что λx — *связывающее определение* (binder) с областью видимости t .) Вхождение x *свободно* (free), если оно находится в позиции, в которой оно не связано никакой вышележащей абстракцией переменной x . Например, вхождения x в $x y$ и $\lambda y. x y$ свободны, а вхождения x в $\lambda x. x$ и $\lambda z. \lambda x. \lambda y. x (y z)$ связаны. В $(\lambda x. x) x$ первое вхождение x связано, а второе свободно.

Терм без свободных переменных называется *замкнутым* (closed); замкнутые термы называют также *комбинаторами* (combinators). Простейший комбинатор, называемый *функцией тождества* (identity function),

```
id = λx. x;
```

не выполняет никаких действий, а просто возвращает свой аргумент.

Операционная семантика

В своей чистой форме лямбда-исчисление не содержит встроенных констант и элементарных операторов — ни чисел, ни арифметических операций, ни условных выражений, ни записей, ни циклов, ни последовательного выполнения выражений, ни ввода-вывода, и т. д. Единственное средство для «вычисления» термов — применение функций к аргументам (которые сами являются функциями). Каждый шаг вычисления состоит в том, что в терме-применении, в котором левый член является абстракцией, связанная переменная в теле этой абстракции заменяется на правый член. Записывается это так:

$$(\lambda x. t_{12}) t_2 \rightarrow [x \mapsto t_2]t_{12}$$

где $[x \mapsto t_2]t_{12}$ означает «терм, получаемый из t_{12} путем замены всех свободных вхождений x на t_2 ». Например, терм $(\lambda x. x) y$ за один шаг вычисления переходит в y , а терм $(\lambda x. x (\lambda x. x)) (u v)$ переходит в $u v$ ($\lambda x. x$). Вслед за Чёрчем, терм вида $(\lambda x. t_{12}) t_2$ называется *редексом* (redex) (reducible expression (redex), «сокращаемое выражение»), а операция переписывания редекса в соответствии с указанным правилом называется *бета-редукцией* (beta-reduction).

В течение многих лет разработчики и теоретики языков программирования изучали различные стратегии вычисления в лямбда-исчислении. Каждая

стратегия определяет, какие редексы в терме могут сработать на следующем шаге вычисления.⁴

- При *полной бета-редукции* (full beta-reduction) в любой момент может сработать любой редекс. На каждом шаге мы выбираем какой-нибудь редекс где-то внутри вычисляемого терма, и проводим шаг редукции. Рассмотрим, например, терм

$$(\lambda x. x) ((\lambda x. x) (\lambda z. (\lambda x. x) z))$$

который можно записать для удобства чтения в виде $\text{id} (\text{id} (\lambda z. \text{id} z))$. В этом терме содержится три редекса:

$$\begin{array}{c} \underline{\text{id} (\text{id} (\lambda z. \text{id} z))} \\ \underline{\text{id} (\text{id} (\lambda z. \text{id} z))} \\ \text{id} (\text{id} (\lambda z. \underline{\text{id} z})) \end{array}$$

При полной бета-редукции можно, например, начать с самого внутреннего редекса, затем обработать промежуточный, а затем — внешний:

$$\begin{array}{l} \text{id} (\text{id} (\lambda z. \underline{\text{id} z})) \\ \rightarrow \text{id} (\underline{\text{id} (\lambda z. z)}) \\ \rightarrow \underline{\text{id} (\lambda z. z)} \\ \rightarrow \lambda z. z \\ \dagger \end{array}$$

- При стратегии *нормального порядка вычислений* (normal order) всегда сначала сокращается самый левый, самый внешний редекс. При такой стратегии указанный терм обрабатывался бы так:

$$\begin{array}{l} \underline{\text{id} (\text{id} (\lambda z. \text{id} z))} \\ \rightarrow \underline{\text{id} (\lambda z. \text{id} z)} \\ \rightarrow \underline{\lambda z. \text{id} z} \\ \rightarrow \lambda z. z \\ \dagger \end{array}$$

При такой стратегии (а также всех перечисленных ниже) отношение вычисления на самом деле является частичной функцией: каждый терм t за шаг переходит не более чем в один терм t' .

- Стратегия *вызыва по имени* (call by name) еще более строга: она не позволяет проводить редукцию внутри абстракций. Начиная с того же самого терма, первые две редукции мы проведем так же, как и при нормальном порядке вычислений, но потом остановимся и будем считать $\lambda z. \text{id} z$ нормальной формой:

$$\begin{array}{l} \underline{\text{id} (\text{id} (\lambda z. \text{id} z))} \\ \rightarrow \underline{\text{id} (\lambda z. \text{id} z)} \\ \rightarrow \lambda z. \text{id} z \\ \dagger \end{array}$$

Варианты вызова по имени использовались в некоторых хорошо известных языках, в том числе в Алголе-60 (Naur et al., 1963) и Haskell

⁴Некоторые исследователи используют термины «редукция» и «вычисление» как синонимы. Другие называют «вычислением» только те стратегии, где какую-то роль имеет понятие «значения», а в остальных случаях говорят о «редукции».

([Hudak et al., 1992](#)). В Haskell, на самом деле, используется оптимизированная версия, известная как *вызов по необходимости* (call by need) ([Wadsworth, 1971; Ariola et al., 1995](#)), в которой вместо того, чтобы пересчитывать аргумент при каждом использовании, при первом вычислении все вхождения аргумента заменяются значением, и таким образом пропадает необходимость вычислять его заново в следующий раз. При такой стратегии требуется во время выполнения совместно использовать структуры данных между представлениями термов — в сущности, получается отношение редукции на *графах* (graphs) абстрактного синтаксиса, а не на синтаксических деревьях.

- В большинстве языков используется стратегия *вызыва по значению* (call by value). В соответствии с ней, сокращаются только самые внешние редексы, и, кроме того, редекс срабатывает только в том случае, если его правая часть уже сведена к *значению* (value) — замкнутому терму, который уже вычислен и не может быть редуцирован далее.⁵ При такой стратегии наш пример будет редуцироваться так:

$$\begin{aligned} &\text{id } \underline{\text{id } (\lambda z. \text{id } z)} \\ \rightarrow &\underline{\text{id } (\lambda z. \text{id } z)} \\ \rightarrow &\lambda z. \text{id } z \\ \vdots & \end{aligned}$$

Стратегия вызова по значению *строга* в том смысле, что аргументы функции всегда вычисляются, независимо от того, используются они в теле функции или нет. С другой стороны, *нестрогие* (non-strict) (или *ленивые*, lazy) стратегии вычисления — вызов по имени или по необходимости, — вычисляют только те аргументы, которые действительно используются.

Выбор стратегии вычисления почти ни на что не влияет в контексте обсуждения систем типов. Вопросы, которые ведут к использованию тех или иных свойств типов, и методы, используемые для ответа на эти вопросы, для всех стратегий практически одинаковы. В этой книге мы используем вызов по значению: во-первых, потому что именно так работает большинство широко известных языков; и, во-вторых, потому что при этом легче всего ввести такие механизмы, как исключения (глава 14) и ссылки (глава 13).

5.2. Программирование на языке лямбда-исчисления

Лямбда-исчисление — значительно более мощный формализм, чем кажется при первом взгляде на его крошечное определение. В этом разделе мы продемонстрируем несколько стандартных примеров программирования в рамках этого формализма. Эти примеры совершенно не означают, что лямбда-

⁵В нашем «фундаментальном» исчислении значениями являются только лямбда-абстракции. В более богатых исчислениях будут присутствовать и другие виды значений: числовые и булевские константы, строки, кортежи значений, записи, состоящие из значений, списки значений и т. п.

исчисление следует считать полноценным языком программирования — во всех распространенных языках те же самые задачи можно решить более понятным и эффективным образом. Скорее, это некоторая разминка, чтобы дать читателю почувствовать, как устроена эта система.

Функции с несколькими аргументами

Заметим для начала, что в лямбда-исчислении отсутствует встроенная поддержка функций с несколькими аргументами. Разумеется, ее было бы нетрудно добавить, однако того же самого результата проще достичь через *функции высшего порядка* (higher-order functions), которые возвращают функции в качестве результата. Допустим, у нас есть терм **s** с двумя свободными переменными **x** и **y**, и мы хотим написать такую функцию **f**, которая выдавала бы для каждой пары аргументов (**v**, **w**) результат подстановки **v** вместо **x** и **w** вместо **y**. Мы пишем не $f = \lambda(x, y). s$, как мы сделали бы это в более богатом языке, а $f = \lambda x. \lambda y. s$. Это означает, что **f** есть функция, которая, получив значение **v** для параметра **x**, выдает функцию, которая, получив значение **w** для параметра **y**, выдает нужный результат. После этого мы поочередно применяем **f** к аргументам, получая запись $f v w$ (т. е., $(f v) w$), которая переходит в $((\lambda y. [x \mapsto v]s) w)$, и далее в $[y \mapsto w][x \mapsto v]s$. Такое преобразование функций с несколькими аргументами в функции высшего порядка называется *каррированием* (currying) в честь Хаскелла Карри, современника Чёрча.

Булевские константы Чёрча

Еще одна языковая конструкция, легко кодируемая в лямбда-исчислении — булевские значения и условные выражения. Определим термы **tru** и **fls** таким образом:

```
tru = λt. λf. t;
fls = λt. λf. f;
```

(Этим термам даны сокращенные имена, чтобы избежать их смешения с элементарными булевскими константами **true** и **false** из главы 3.)

Можно считать, что термы **tru** и **fls** представляют собой булевские значения «истина» и «ложь» в том смысле, что с их помощью мы можем выполнить операцию проверки булевского значения на истинность. А именно, мы можем определить комбинатор **test**, такой, что **test b v w** переходит в **v**, если **b** равно **tru**, и в **w**, если **b** равно **fls**.

```
test = λl. λm. λn. l m n;
```

Комбинатор **test** почти ничего не делает: **test b v w** просто переходит в **b v w**. В сущности, булевские значения являются условными выражениями: они принимают два аргумента и выбирает из них либо первый (если это **tru**), либо второй (если это **fls**). Например, терм **test tru v w** редуцируется таким образом:

```

test tru v w
= (λl. λm. λn. l m b) tru v w по определению
→ (λm. λn. tru m b) v w редукция подчеркнутого выражения
→ (λn. tru v b) w редукция подчеркнутого выражения
→ tru v w редукция подчеркнутого выражения
= (λt. λf. t) v w по определению
→ (λf. v) w редукция подчеркнутого выражения
→ v редукция подчеркнутого выражения

```

Несложно также определить в виде функций такие булевские операторы, как логическая конъюнкция:

```
and = λb. λc. b c fls;
```

То есть, `and` — это функция, которая, получив два булевских значения `b` и `c`, возвращает `c`, если `b` равно `tru` и `fls`, если `b` равно `fls`; таким образом, `and b c` выдает `tru`, если и `b`, и `c` равны `tru`, и `fls`, если либо `b`, либо `c` окажутся равными `fls`.

```

and tru tru;
⇒ (λt. λf. t)

and tru fls;
⇒ (λt. λf. f)

```

Упражнение 5.2.1 [★]: Определите логические функции `or` («или») и `not` («не»).

Пары

При помощи булевских констант мы можем закодировать пары значений в виде термов:

```

pair = λf. λs. λb. b f s;
fst = λp. p tru;
snd = λp. p fls;

```

Это означает, что `pair v w` — функция, которая, будучи применена к булевскому значению `b`, применяет `b` к `v` и `w`. По определению булевских констант, при таком вызове получится `v`, если `b` равняется `tru`, и `w`, если `b` равняется `fls`, так что функции первой и второй проекции `fst` и `snd` можно получить, просто подав в пару соответствующие булевские значения. Вот как проверить, что `fst (pair v w) →* v`:

```

fst (pair v w)
= fst ((λf. λs. λb. b f s) v w) по определению
→ fst ((λs. λb. b v s) w) редукция подчеркнутого выражения
→ fst (λb. b v w) редукция подчеркнутого выражения
= (λp. p tru) (λb. b v w) по определению
→ (λb. b v w) tru редукция подчеркнутого выражения
→ tru v w редукция подчеркнутого выражения
→* v как показано ранее.

```

Числа Чёрча

После всего, что мы видели, представление чисел в виде лямбда-термов будет лишь ненамного сложней. *Числа Чёрча* (Church numerals) c_0, c_1, c_2 , и т. д. можно определить таким образом:

```

c₀ = λs. λz. z;
c₁ = λs. λz. s z;
c₂ = λs. λz. s (s z);
c₃ = λs. λz. s (s (s z));

```

и т. д. Здесь каждое число n представляется комбинатором c_n , который принимает два аргумента, s и z («функцию следования» и «ноль»), и n раз применяет s к z . Как и в случае с булевскими константами и парами, такое кодирование превращает числа в активные сущности: число n представляется в виде функции, которая что-то делает n раз — своего рода активное число по основанию 1.

(Читатель мог уже заметить, что c_0 и fls записываются одним и тем же термом. Такие «каламбуры» часто встречаются в языках ассемблера, где одна и та же комбинация битов может представлять собой множество разных значений — целое число, число с плавающей точкой, адрес, четыре символа и т. п., — в зависимости от того, как интерпретируются биты. Аналогичная ситуация в низкоуровневых языках вроде C, где 0 и `false` тоже представляются одинаково.)

Функцию следования на числах Чёрча можно определить так:

```
scc = λn. λs. λz. s (n s z);
```

Терм scc — это комбинатор, который принимает число Чёрча n и возвращает другое число Чёрча, — то есть, возвращает функцию, которая принимает аргументы s и z , и многократно применяет s к z . Нужное число применений s к z мы получаем, сначала передав s и z в качестве аргументов n , а затем явным образом применив s еще раз к результату.

Упражнение 5.2.2 [★★]: Найдите ещё один способ определения функции следования на числах Чёрча.

Похожим образом, сложение на числах Чёрча можно осуществлять с помощью терма $plus$, который принимает в качестве аргументов два числа Чёрча, m и n , и возвращает еще одно число Чёрча — т. е., функцию, которая принимает аргументы s и z , применяет s к z n раз (передавая s и z в качестве аргументов n), а потом применяет s еще m раз к результату:

```
plus = λm. λn. λs. λz. m s (n s z);
```

Для реализации умножения используется еще один трюк: поскольку `plus` принимает аргументы по одному, применение его к одному аргументу `n` дает функцию, которая добавляет `n` к любому данному ей аргументу. Можно передать эту функцию в качестве первого аргумента `m`, а в качестве второго дать `c0`, и это будет означать «применить функцию, добавляющую `n` к своему аргументу, к нулю и повторить `m` раз», т. е., «сложить `m` копий числа `n`».

```
times = λm. λn. m (plus n) c0;
```

Упражнение 5.2.3 [★★]: *Можно ли определить умножение на числах Чёрча без использования `plus`?*

Упражнение 5.2.4 [РЕКОМЕНДУЕТСЯ, ★★]: *Определите терм для возведения чисел в степень.*

Чтобы проверить, является ли число Чёрча нулем, нужно найти какую-то подходящую для этой цели пару аргументов, — а именно, нужно применить указанное число к паре термов `zz` и `ss`, таких чтобы применение `ss` к `zz` один или более раз давало `f1s`, а отсутствие применения давало `tru`. Понятно, что в качестве `zz` нужно просто взять `tru`. Для `ss` же мы используем функцию, которая игнорирует свой аргумент и всегда возвращает `f1s`:

```
iszro = λm. m (λx. f1s) tru;

iszro c1;
> (λt. λf. f)
iszro (times c0 c2);
> (λt. λf. t)
```

Как ни странно, определить вычитание на числах Чёрча намного сложнее, чем сложение. Для этого можно воспользоваться следующей довольно хитрой «функцией предшествования», которая возвращает `c0`, если передать в качестве аргумента `c0`, и возвращает `ci`, если передать в качестве аргумента `ci+1`:

```
zz = pair c0 c0;
ss = λp. pair (snd p) (plus c1 (snd p));
prd = λm. fst (m ss zz);
```

Это определение работает так: мы используем `m` в качестве функции и применяем с ее помощью `m` копий функции `ss` к начальному значению `zz`. Каждая копия `ss` принимает в качестве аргумента пару чисел `pair ci cj` и выдает в результате пару `pair cj cj+1` (см. рис. 5.1). Таким образом, при `m`-кратном применении `ss` к `pair c0 c0` получается `pair c0 c0`, если `m = 0`, и `pair cm-1 cm` при положительном `m`. В обоих случаях, в первом компоненте пары находится искомый предшественник.

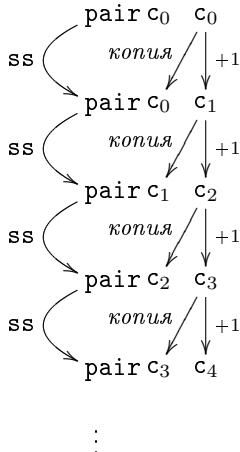


Рис. 5.1. «Внутренний цикл» функции предшествования.

Упражнение 5.2.5 [★★]: Определите функцию вычитания при помощи *prd*.

Упражнение 5.2.6 [★★]: Сколько примерно шагов вычисления (в зависимости от *n*) требуется, чтобы получить *prd c_n*?

Упражнение 5.2.7 [★★]: Напишите функцию *equal*, которая проверяет два числа на равенство и возвращает Чёрчеву булевскую константу. Например:

```

equal c3 c3
=> (λt. λf. t)

equal c3 c2
=> (λt. λf. f)
  
```

Подобными же методами можно определить другие распространенные типы данных: списки, деревья, массивы, записи с вариантами, и т. п.

Упражнение 5.2.8 [РЕКОМЕНДУЕТСЯ, ★★★]: Список можно представить в лямбда-исчислении через его функцию свертки *fold*. (В OCaml эта функция называется *fold_left*; ее иногда еще называют *reduce*.) Например, список *[x, y, z]* становится функцией, которая принимает два аргумента *s* и *n*, и возвращает *s x (s y (s z n))*. Как будет выглядеть представление *nil*? Напишите функцию *cons*, которая принимает элемент *h* и список (то есть, функцию свертки) *t*, и возвращает подобное представление списка, полученного добавлением *h* в голову *t*. Напишите функции *isnil* и *head*, каждая из которых принимает список в качестве параметра. Наконец, напишите функцию *tail* для такого представления списков (это намного сложнее; придется использовать трюк, аналогичный тому, что использовался при определении *prd* для чисел).

Расширенное исчисление

Мы убедились, что булевские значения, числа и операции над ними могут быть закодированы средствами чистого лямбда-исчисления. Строго говоря, все нужные нам программы мы можем писать, не выходя за рамки этой системы. Однако при работе с примерами часто бывает удобно включить в нее элементарные булевские значения и числа (а может быть, и другие типы данных). Если нам нужно совершенно точно указать, с какой системой мы в данный момент работаем, то для чистого лямбда-исчисления, определяемого на рис. 5.3, мы будем использовать обозначение λ , а для системы, в которую добавлены булевские и арифметические выражения с рис. 3.1 и 3.2 — обозначение λNB .

В λNB по сути есть две разные реализации булевских значений и две реализации чисел: настоящие и закодированные методом, описанным в этой главе. Мы можем выбирать между ними при написании программ. Разумеется, эти две реализации нетрудно преобразовать друг в друга. Чтобы перевести булевское значение по Чёрчу в элементарное булевское значение, нужно применить его к значениям `true` и `false`:

```
realbool = λb. b true false;
```

Для обратного преобразования используется условное выражение:

```
churchbool = λb. if b then tru else fls;
```

Можно встроить эти преобразования в операции высшего порядка. Вот проверка на равенство для чисел Чёрча, возвращающая настоящее логическое значение:

```
realeq = λm. λn. (equal m n) true false;
```

Аналогично мы можем преобразовать число Чёрча в соответствующее элементарное число, применив его к `succ` и 0:

```
realnat = λm. m (λx. succ x) 0;
```

Мы не можем напрямую применить `m` к `succ`, поскольку сама по себе запись `succ` не имеет синтаксического смысла: мы определили арифметические выражения так, что `succ` всегда должен к чему-то применяться. Это требование мы обходим, обернув `succ` в маленькую функцию, которая всегда возвращает `succ` от своего аргумента.

Причины, по которым элементарные булевские и числовые значения оказываются полезны при работе с примерами, в основном связаны с порядком вычислений. Рассмотрим, например, терм `scc c1`. Исходя из приведенного выше обсуждения, мы могли бы ожидать, что он должен при вычислении давать число Чёрча `c2`. На самом деле этого не происходит:

```
scc c1;
▷ (λs. λz. s ((λs'. λz'. s' z') s z))
```

Этот терм содержит в себе редекс, который при вычислении привел бы нас (за два шага) к c_2 , однако согласно правилам вызова по значению мы ещё не можем сделать это, поскольку редекс находится внутри лямбда-абстракции.

Никакой фундаментальной проблемы здесь нет: терм, получающийся при вычислении $scc c_1$, очевидным образом *поведенчески эквивалентен* (behaviorally equivalent) c_2 , в том смысле, что применение этого терма к паре аргументов v и w всегда даст тот же результат, что и применение c_2 к тем же аргументам. Однако, необходимость доделать вычисление затрудняет проверку того, что наша функция scc ведет себя как надо. В случае более сложных арифметических вычислений трудность еще возрастает. Например, $\text{times } c_2 c_2$ дает в результате не c_4 , а такое чудовищное выражение:

```
times c2 c2;

⇒ (λs .
    λz .
      (λs' . λz' . s' (s' z')) s
      ((λs' .
          λz' .
            (λs'' . λz'' . s'' (s'' z'')) s'
            (((λs'' . λz'' . z'') s' z')) s
            z))
```

Можно убедиться, что этот терм ведет себя так же, как c_4 , с помощью проверки на равенство:

```
equal c4 (times c2 c2);

⇒ (λt . λf . t)
```

Однако более прямой способ — взять $\text{times } c_2 c_2$ и преобразовать в элементарное число:

```
realnat (times c2 c2);

⇒ 4
```

Функция преобразования передает выражению $\text{times } c_2 c_2$ два дополнительных аргумента, которых оно ожидает, и заставляет выполнить все задержанные вычисления в его теле.

Рекурсия

Вспомним, что терм, который не может продвинуться дальше согласно отношению вычисления, называется *нормальной формой* (normal form). Любопытно, что некоторые термы не могут быть вычислены до нормальной формы. Например, *расходящийся комбинатор* (divergent combinator)

```
omega = (λx . x x) (λx . x x);
```

содержит только один редекс, но шаг вычисления этого редекса дает в результате опять *omega!* Про термы, не имеющие нормальной формы, говорят, что они *расходятся* (diverge).

Комбинатор `omega` можно обобщить до полезного терма, который называется *комбинатором неподвижной точки* (fixed-point combinator),⁶ с помощью которого можно определять рекурсивные функции, например, `factorial`.⁷

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));
```

Как и `omega`, комбинатор `fix` имеет сложную структуру с повторами; глядя на определение, трудно понять, как он работает. Вероятно, получить интуитивное представление о его поведении удобнее всего, рассмотрев его действие на конкретном примере.⁸ Допустим, мы хотим написать рекурсивное определение функции вида $h = \langle\text{тело, содержащее } h\rangle$ — т. е., построить такое определение, в котором правая часть использует саму функцию, которую мы определяем (например, как в определении факториала на с. 72). Идея состоит в том, чтобы рекурсивное определение «разворачивалось» там, где оно встречается. Например, факториалу интуитивно соответствует определение:

```
if n=0 then 1
else n * (if n-1=0 then 1
            else (n-1) * (if n-2=0 then 1
                            else (n-2) * ...))
```

или, в терминах чисел Чёрча,

```
if realeq n c0 then c1
else times n (if realeq (prd n) c0 then c1
                  else times (prd n)
                        (if realeq (prd (prd n)) c0 then c1
                            else times (prd (prd n)) ...))
```

Такого эффекта можно добиться при помощи комбинатора `fix`, сначала определив $g = \langle\text{тело, содержащее } f\rangle$, а затем $h = \text{fix } g$. Например, функцию факториала можно определить через

```
g = λfct. λn. if realeq n c0 then c1 else (times n (fct (prd n)));
factorial = fix g;
```

На рис. 5.2 показано, что происходит при вычислении с термом `factorial c3`. Ключевое свойство, которое обеспечивает работу этого вычисления, — это $fct \rightarrow^* g \ fct \ n$. Таким образом, `fct` — своего рода «самовоспроизводящийся автомат», который, будучи применен к аргументу `n`, передает *самого себя* и `n` в качестве аргументов `g`. Там, где первый аргумент встречается в теле `g`, мы получим еще одну копию `fct`, которая, будучи применена к аргументу, опять передаст *самое себя* и аргумент внутрь `g`, и т. д. При каждом рекурсивном вызове с помощью `fct` мы разворачиваем очередную копию `g` и снабжаем ее очередными копиями `fct`, готовыми развернуться еще дальше.

⁶Его также часто называют *Y-комбинатором с вызовом по значению* (call-by-value Y-combinator). Плоткин (Plotkin, 1975) использовал обозначение `Z`.

⁷Заметим, что более простой комбинатор неподвижной точки с вызовом по имени:

```
Y = λf. (λx. f (x x)) (λx. f (x x))
```

при использовании вызова по значению бесполезен, поскольку при любом `g` выражение `Y g` расходится.

⁸Определение `fix` также можно вывести непосредственно из базовых принципов (например, в Friedman and Felleisen, 1996, глава 9), однако такой вывод тоже достаточно хитроумен.

```

factorial c3
=   fix g c3
→   h h c3
    где h = λx. g (λy. x x y)
→   g fct c3
    где fct = λy. h h y
→   (λn. if realeq n c0
        then c1
        else times n (fct (prd n)))
        c3
→   if realeq c3 c0
        then c1
        else times c3 (fct (prd c3))
→* times c3 (fct (prd c3))
→* times c3 (fct c'2)
    где c'2 поведенчески эквивалентен c2
→* times c3 (g fct c'2)
→* times c3 (times c'2 (g fct c'1))
    где c'1 поведенчески эквивалентен c1
    (те же шаги повторяются для g fct c'2)
→* times c3 (times c'2 (times c'1 (g fct c'0)))
    где c'0 поведенчески эквивалентен c0
    (аналогично)
→* times c3 (times c'2 (times c'1 (if realeq c'0 c0 then c1
        else ...)))
→* times c3 (times c'2 (times c'1 c1))
→* c'6
    где c'6 поведенчески эквивалентен c6

```

Рис. 5.2. Вычисление factorial c₃

Упражнение 5.2.9 [★]: Почему в определении *g* мы использовали элементарную формулу *if*, а не функцию *test*, работающую с Чёрчевыми булевскими значениями? Покажите, как определить функцию *factorial* при помощи *test* вместо *if*.

Упражнение 5.2.10 [★★]: Напишите функцию *churchnat*, переводящую элементарное натуральное число в представление Чёрча.

Упражнение 5.2.11 [РЕКОМЕНДУЕТСЯ, ★★]: При помощи *fix* и кодирования списков из упражнения 5.2.8 напишите функцию, суммирующую список, состоящий из чисел Чёрча.

Представление

Прежде чем закончить рассмотрение примеров и заняться формальным определением лямбда-исчисления, следует задаться еще одним, последним во-

просом: что, строго говоря, означает утверждение, что числа Чёрча *представляют* обыкновенные числа?

Чтобы ответить на этот вопрос, вспомним, что такое обыкновенные числа. Существует много (эквивалентных) определений; в этой книге мы выбрали такое (рис. 3.2):

- константа 0,
- операция `iszzero`, отображающая числа на булевские значения, и
- две операции, `succ` и `pred`, отображающие числа на числа.

Поведение арифметических операций определяется правилами вычисления из рис. 3.2. Эти правила говорят нам, например, что 3 следует за 2, и что `iszzero 0` истинно.

Кодирование по Чёрчу представляет каждый из этих элементов в виде лямбда-терма (то есть, функции):

- Терм `c0` представляет число 0.

Как мы видели на с. 83, имеются также «неканонические представления» чисел в виде термов. Например, терм $\lambda s. \lambda z. (\lambda x. x) z$, который поведенчески эквивалентен `c0`, также представляет число 0.

- Термы `scc` и `prd` представляют арифметические операции `succ` и `pred`, в том смысле, что, если `t` является представлением числа `n`, то `scc t` дает при вычислении представление числа `n + 1`, а `prd t` дает представление `n - 1` (или 0, если `n` равно 0).
- Терм `iszro` представляет операцию `iszzero`, в том смысле, что, если `t` является представлением 0, то `iszro t` дает при вычислении `true`,⁹ а если `t` представляет ненулевое число, то `iszro t` дает `false`.

Учитывая всё вышеизложенное, представим, что у нас есть программа, которая проделывает некоторые сложные численные вычисления и выдает булевский результат. Если мы заменим все числа и арифметические операции лямбда-термами, которые их представляют, и запустим получившуюся программу, мы получим тот же самый результат. Таким образом, с точки зрения окончательного результата программ, нет никакой разницы между настоящими числами и их представлениями по Чёрчу.

5.3. Формальности

В оставшейся части главы мы даем точное определение синтаксиса и операционной семантики лямбда-исчисления. По большей части, всё устроено так же, как в главе 3 (чтобы не повторять всё заново, мы здесь определяем только чистое лямбда-исчисление, без булевских значений и чисел). Однако операция подстановки терма вместо переменной связана с неожиданными сложностями.

⁹Строго говоря, по нашему определению, `iszro t` вычисляется в *представление* `true` в виде терма, но для простоты обсуждения мы забудем про это различие. Можно аналогичным образом выстроить объяснение того, как именно Чёрчевые булевские константы представляют настоящие булевские значения.

Синтаксис

Как и в главе 3, абстрактную грамматику, определяющую термы (на с. 73) следует рассматривать как сокращенную запись индуктивно определенного множества абстрактных синтаксических деревьев.

Определение 5.3.1 [Термы]: Пусть имеется счетное множество имен переменных \mathcal{V} . Множество термов — это наименьшее множество \mathcal{T} такое, что

1. $x \in \mathcal{T}$ для всех $x \in \mathcal{V}$;
2. Если $t_1 \in \mathcal{T}$ и $x \in \mathcal{V}$, то $\lambda x. t_1 \in \mathcal{T}$;
3. Если $t_1 \in \mathcal{T}$ и $t_2 \in \mathcal{T}$, то $t_1 t_2 \in \mathcal{T}$.

Размер (size) терма t можно определить точно так же, как мы это сделали для арифметических выражений в определении 3.3.2. Интереснее тот факт, что можно дать простое индуктивное определение множества свободных переменных, встречающихся в терме.

Определение 5.3.2 Множество свободных переменных (*free variables*) терма t записывается как $FV(t)$ и определяется так:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x. t_1) &= FV(t_1) \setminus \{x\} \\ FV(t_1 t_2) &= FV(t_1) \cup FV(t_2) \end{aligned}$$

Упражнение 5.3.3 [★★]: Постройте строгое доказательство утверждения: $|FV(t)| \leq \text{size}(t)$ для любого терма t .

Подстановка

Операция подстановки при подробном рассмотрении оказывается довольно непростой. В этой книге мы будем использовать два разных определения, каждое из которых удобно для своих целей. В этом разделе мы введем первое из них, краткое и интуитивно понятное. Оно хорошо работает в примерах, в математических определениях и доказательствах. Второе, рассматриваемое в главе 6, использует более сложную нотацию и зависит от альтернативного «представления де Брауна» для термов, где именованные переменные заменяются на числовые индексы. Это представление оказывается более удобным для конкретных реализаций на ML, которые обсуждаются в последующих главах.

Поучительно прийти к определению подстановки, сделав пару неудачных попыток. Проверим сначала самое наивное рекурсивное определение. (С формальной точки зрения, мы определяем функцию $[x \mapsto s]$ индукцией по аргументу t .):

$$\begin{aligned} [x \mapsto s]x &= s \\ [x \mapsto s]y &= y && \text{если } x \neq y \\ [x \mapsto s](\lambda y. t_1) &= \lambda y. [x \mapsto s]t_1 \\ [x \mapsto s](t_1 t_2) &= ([x \mapsto s]t_1) ([x \mapsto s]t_2) \end{aligned}$$

Такое определение в большинстве случаев работает правильно. Например, оно дает

$$[x \mapsto (\lambda z. z w)](\lambda y. x) = \lambda y. \lambda z. z w$$

что соответствует нашему интуитивному представлению о том, как должна себя вести подстановка. Однако при неудачном выборе имен связанных переменных это определение не работает. Например:

$$[x \mapsto y](\lambda x. x) = \lambda x. y$$

Это противоречит базовой интуитивной идеи функциональной абстракции: *имена связанных переменных не должны ни на что влиять* — функция тождества остается самой собой, будь она записана в виде $\lambda x. x$, $\lambda y. y$ или $\lambda franz. franz$. Если эти термы ведут себя по-разному при подстановке, они поведут себя по-разному и при редукции, а это явно неправильно.

Очевидно, что первая ошибка, которую мы допустили в наивном определении подстановки, состоит в том, что мы не отличали *свободное* вхождение переменной x в терм t (которое при подстановке нужно заменять) от связанныго (которое заменять не нужно). Когда мы доходим до абстракции, связывающей имя x внутри t , операция подстановки должна останавливаться. Предпримем следующую попытку:

$$\begin{aligned} [x \mapsto s]x &= s \\ [x \mapsto s]y &= y && \text{если } y \neq x \\ [x \mapsto s](\lambda y. t_1) &= \begin{cases} \lambda y. t_1 & \text{если } y = x \\ \lambda y. [x \mapsto s]t_1 & \text{если } y \neq x \end{cases} \\ [x \mapsto s](t_1 t_2) &= ([x \mapsto s]t_1) ([x \mapsto s]t_2) \end{aligned}$$

Это уже лучше, но все-таки еще не вполне правильно. Посмотрим, например, что получается, когда мы пытаемся подставить терм z вместо переменной x в терме $\lambda z. x$:

$$[x \mapsto z](\lambda z. x) = \lambda z. z$$

В этот раз мы совершили, в сущности, противоположную ошибку: превратили функцию-константу $\lambda z. x$ в функцию тождества! Это снова случилось оттого, что мы выбрали z в качестве имени связанной переменной в функции-константе, так что что-то мы до сих пор делаем не так.

Ситуация, в которой свободные переменные терма s становятся связанными при их наивной подстановке в терм t , называется *захватом переменных* (variable capture). Чтобы избежать его, нужно убедиться в том, что имена связанных переменных в t отличаются от имен свободных переменных в s . Операция подстановки, которая работает именно так, называется *подстановкой, свободной от захвата* (capture-avoiding substitution). (Обычно, когда просто говорят «подстановка», именно такую подстановку и имеют в виду.) Мы можем добиться требуемого эффекта, если добавим ко второму варианту еще одно условие при подстановке в терм-абстракцию:

$$\begin{aligned} [x \mapsto s]x &= s \\ [x \mapsto s]y &= y && \text{если } y \neq x \\ [x \mapsto s](\lambda y. t_1) &= \begin{cases} \lambda y. t_1 & \text{если } y = x \\ \lambda y. [x \mapsto s]t_1 & \text{если } y \neq x \text{ и } y \notin FV(s) \end{cases} \\ [x \mapsto s](t_1 t_2) &= ([x \mapsto s]t_1) ([x \mapsto s]t_2) \end{aligned}$$

Теперь почти все правильно: наше определение подстановки делает то, что требуется, *когда оно вообще что-то делает*. Проблема заключается в том, что последнее изменение превратило подстановку из полной функции в частичную. Например, новое определение не выдает никакого результата для $[x \mapsto y z](\lambda y. x y)$: связанная переменная y терма, в который производится подстановка, не равна x , но она встречается как свободная в терме $(y z)$, и ни одна строка определения к ней не применима.

Распространенное решение этой проблемы в литературе по системам типов и лямбда-исчислению состоит в том, что термы рассматриваются «с точностью до переименования переменных». (Чёрч называл операцию последовательного переименования переменных в терме *альфа-конверсией* (alpha-conversion). Этот термин употребляется и до сих пор — мы могли бы сказать, что рассматриваем термы «с точностью до альфа-конверсии».)

Соглашение 5.3.4 *Термы, отличающиеся только именами связанных переменных, взаимозаменимы во всех контекстах.*

На практике это означает, что имя любой λ -связанной переменной можно заменить на другое (последовательно проводя это переименование в теле λ) всегда, когда это оказывается удобным. Например, если мы хотим вычислить $[x \mapsto y z](\lambda y. x y)$, мы сначала переписываем $(\lambda y. x y)$ в виде, скажем, $(\lambda w. x w)$. Затем мы вычисляем $[x \mapsto y z](\lambda w. x w)$, что дает нам $(\lambda w. y z w)$.

Это соглашение делает наше определение «практически полным», поскольку каждый раз, как мы пытаемся его применить к аргументам, к которым оно неприменимо, мы можем исправить дело переименованием, так, чтобы все условия выполнялись. В сущности, приняв это соглашение, мы можем сформулировать определение подстановки чуть короче. Мы можем отбросить первый вариант в определении для абстракций, поскольку всегда можно предположить (применяя, если надо, переименование), что связанная переменная y отличается как от x , так и от свободных переменных s . Определение принимает окончательный вид.

Определение 5.3.5 [Подстановка]:

$$\begin{aligned} [x \mapsto s]x &= s \\ [x \mapsto s]y &= y && \text{если } y \neq x \\ [x \mapsto s](\lambda y. t_1) &= \lambda y. [x \mapsto s]t_1 && \text{если } y \neq x \text{ и } y \notin FV(s) \\ [x \mapsto s](t_1 t_2) &= ([x \mapsto s]t_1) ([x \mapsto s]t_2) \end{aligned}$$

Операционная семантика

Операционная семантика лямбда-термов вкратце представлена на рис. 5.3. Множество значений в этом исчислении более интересно, чем было в случае арифметических выражений. Поскольку вычисление (с вызовом по значению) останавливается, когда достигает лямбды, значениями являются произвольные лямбда-термы.

Отношение вычисления показано в правом столбце рисунка. Как и в случае арифметических выражений, имеется два типа правил: *рабочее правило E-APPABS* и *правила соответствия E-APP1* и *E-APP2*.

→ (бестиповое)

Синтаксис	термы:	Вычисление
$t ::=$	x $\lambda x. t$ $t \ t$	$t \rightarrow t'$
$v ::=$	$\lambda x. t$	$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2}$ (E-APP1) $\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2}$ (E-APP2)
	значения: значение-абстракция	$(\lambda x. t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12}$ (E-APPABS)

Рис. 5.3. Бестиповое лямбда-исчисление (λ)

Обратите внимание, как выбор метапеременных в этих правилах помогает управлять порядком вычислений. Поскольку v_2 может относиться только к значениям, левая сторона правила E-APPABS соответствует всем применением термов, в которых терм-аргумент является значением. Точно так же, E-APP1 относится к тем применениям, в которых левая часть *не является* значением, поскольку t_1 может обозначать любой терм, но предпосылка правила требует, чтобы t_1 мог совершить шаг вычисления. Напротив, E-APP2 не срабатывает, пока левая часть не станет значением, которое может быть обозначено метапеременной v . Совместно эти правила полностью определяют порядок вычисления терма вида $t_1 \ t_2$: сначала работает E-APP1, пока t_1 не сведется к значению, затем E-APP2 применяется до тех пор, пока t_2 не окажется значением, и, наконец, само правило E-APPABS производит само применение.

Упражнение 5.3.6 [★★]:

Модифицируйте эти правила так, чтобы описать три другие стратегии вычисления: полную бета-редукцию, нормальный порядок и ленивое вычисление.

Заметим, что в чистом лямбда-исчислении единственны возможные значения — это лямбда-абстракции, так что, если E-APP1 доводит t_1 до значения, это значение должно быть лямбда-абстракцией. Разумеется, это утверждение перестает быть справедливым, как только мы добавляем в язык другие конструкции, скажем, элементарные булевские значения, поскольку при этом у нас появляются новые виды значений.

Упражнение 5.3.7 [★★, →]:

В упражнении 3.5.16 дается альтернативное представление операционной семантики булевых и арифметических выражений, в котором тупиковые термы дают при вычислении особую константу *wrong*. Распространите эту семантику на λNB .

Упражнение 5.3.8 [★★]:

В упражнении 3.5.17 вводится стиль вычисления арифметических выражений «с большим шагом», в котором базовое отношение вычисления означает «терм t при вычислении дает окончательный результат v ». Покажите, как сформулировать правила вычисления лямбда-термов в этом стиле.

5.4. Дополнительные замечания

Бестиповое лямбда-исчисление было разработано Чёрчем и его коллегами в 20-е и 30-е годы (Church, 1941). Основополагающий труд по всем вопросам бестипового лямбда-исчисления — книга Барендргета (Barendregt, 1984); работа Хиндли и Селдина (Hindley and Seldin, 1986) уже по охвату, но легче для чтения. Статья Барендргета (Barendregt, 1990) в «Справочнике по теоретической информатике» представляет собой краткий обзор. Сведения о лямбда-исчислении можно найти также во множестве учебников по функциональным языкам программирования (Abelson and Sussman, 1985; Friedman, Wand, and Haynes, 2001; Peyton Jones and Lester, 1992) и по семантике языков программирования (напр., Schmidt, 1986; Gunter, 1992; Winskel, 1993; Mitchell, 1996). Систематический метод кодирования различных структур данных в виде лямбда-термов описан в статье Бёма и Берардуччи (Böhm and Berarducci, 1985).

Несмотря на название, Карри не считал себя автором идеи каррирования. Её открытие обычно приписывают Шёнфинкелю (Schönfinkel, 1924), однако основная идея еще в XIX веке была известна некоторым математикам, в том числе Фреге и Кантору.

Возможно, у этой системы
 найдутся приложения не только
 в роли логического исчисления.

Алонсо Чёрч, 1932

Глава 6

Представление термов без использования имен

В предыдущей главе мы работали с термами «с точностью до переименования связанных переменных». Мы договорились, что связанные переменные можно в любой момент переименовать, чтобы провести подстановку или если новое имя по каким-то причинам удобнее. В сущности, «внешний вид» имени связанной переменной может быть любым. Такое соглашение отлично работает при обсуждении основных идей лямбда-исчисления и помогает понятно записывать доказательства. Однако, при реализации исчисления в программе нам нужно иметь единое представление для каждого терма; в частности, требуется решить, как будут представлены вхождения переменных. Есть несколько способов решить эту задачу:

1. Можно представлять переменные символически, как мы это делали до сих пор, однако вместо неявного переименования мы при необходимости явно заменяем при подстановке связанные переменные «свежими» именами, чтобы избежать захвата.
2. Можно представлять переменные символически, но потребовать, чтобы имена всех связанных переменных отличались друг от друга и от всех где-либо встречающихся свободных переменных. Такое соглашение (иногда его называют *соглашение Барендргета*, Barendregt convention) более строго, чем наше, поскольку не разрешает переименовывать переменные «на ходу» в произвольные моменты. Однако это правило не безопасно относительно подстановки (или бета-редукции): поскольку подставляемый терм копируется, то нетрудно построить примеры, где в результате подстановки получается терм, в котором у нескольких λ -абстракций будет одно и то же имя связанной переменной. Следовательно, после каждого шага вычисления, включающего подстановку, должен следовать шаг переименования, восстанавливающий инвариант.

В этой главе изучается бестиповое лямбда-исчисление, λ (рис. 5.3). Соответствующая реализация на OCaml называется `fulluntyped`.

3. Можно сконструировать некоторое «каноническое» представление переменных и термов, при котором переименование не нужно.
4. Можно вообще избежать понятия подстановки с помощью механизмов вроде *явных подстановок* (explicit substitutions) (Abadi, Cardelli, Curien, and Lévy, 1991a).
5. Можно избежать использования *переменных* (variables), если работать в языке, основанном на комбинаторах, например, *комбинаторной логике* (combinatory logic) (Curry and Feys, 1958; Barendregt, 1984) — варианте лямбда-исчисления, в котором вместо процедурной абстракции используются комбинаторы, — или на языке Бэкуса FP (Backus, 1978).

У каждой из этих схем есть свои сторонники, и выбор между ними — до некоторой степени дело вкуса (в серьезных реализациях компиляторов следует также учитывать соображения производительности, но нас они сейчас не волнуют). Мы выбираем третий вариант, который, по нашему опыту, будет лучше масштабироваться, когда потребуется работать с некоторыми более сложными интерпретаторами из этой книги. Его преимущество в том, что при ошибках реализации алгоритм сразу же выдаёт очевидно ошибочные результаты в самых простых случаях. Это позволяет достаточно быстро обнаруживать и исправлять ошибки. Напротив, известны случаи, когда в реализациях, основанных на именованных переменных, ошибки обнаруживались спустя месяцы и годы. Наша реализация использует хорошо известный метод, изобретенный Николасом де Брауном (de Bruijn, 1972).

6.1. Термы и контексты

Идея де Брауна состояла в том, чтобы представлять термы более естественным — хотя и более трудным для чтения — образом, обеспечив во вхождениях переменных *прямые указания* на их связывающие определения, вместо того, чтобы называть их по имени. Для этого можно заменить именованные переменные натуральными числами так, чтобы число k означало «переменная, связанная k -й охватывающей λ ». Например, обычный терм $\lambda x. x$ соответствует *безымянному терму* (nameless term) $\lambda. 0$, а терму $\lambda x. \lambda y. x$ (у x) соответствует $\lambda. \lambda. 1$ (0 1). Безымянные термы иногда еще называют *термами де Брауна* (de Bruijn terms), а нумерованные переменные в них называются *индексами де Брауна* (de Bruijn indices).^{*} Разработчики компиляторов используют для этого понятия термин «статические расстояния».

Упражнение 6.1.1 [★]: Для каждого из следующих комбинаторов

```
c0 = λs. λz. z;
c2 = λs. λz. s (s z);
plus = λm. λn. λs. λz. m s (n s z);
fix = λf. (λx. f (λy. (x x) y)) (λx. f (λy. (x x) y));
foo = (λx. (λx. x)) (λx. x);
```

*Фамилия de Bruijn читается «де Браун». В русскоязычной литературе встречаются варианты транслитерации «де Брейн», «де Брюн» и «де Брайн». — прим. перев.

запишите соответствующий безымянный терм.

Формально мы определяем синтаксис безымянных термов почти так же, как определялся синтаксис обыкновенных термов (5.3). Единственное различие состоит в том, что требуется внимательно следить, сколько свободных переменных может содержать каждый терм. То есть, требуется различать множества термов без свободных переменных (которые называются *0-термами*, 0-terms), термов, в которых есть максимум одна свободная переменная (1-термы), и так далее.

Определение 6.1.2 [ТЕРМЫ]: Пусть \mathcal{T} — наименьшее семейство множеств $\{\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \dots\}$, такое, что

1. $k \in \mathcal{T}_n$, если $0 \leq k < n$;
2. если $t_1 \in \mathcal{T}_n$ и $n > 0$, то $\lambda.t_1 \in \mathcal{T}_{n-1}$;
3. если $t_1 \in \mathcal{T}_n$ и $t_2 \in \mathcal{T}_n$, то $(t_1 \ t_2) \in \mathcal{T}_n$.

(Заметим, что мы имеем здесь стандартное индуктивное определение, но определяем семейство множеств, индексируемое числами, а не одно множество.) Элементы каждого множества \mathcal{T}_n называются *n-термами*.

Элементы \mathcal{T}_n — это термы с *не более*, чем n свободными, пронумерованными от 0 до $n - 1$: каждый элемент \mathcal{T}_n не обязан содержать свободные переменные со всеми этими номерами, да и вообще не обязан иметь какие-либо свободные переменные. В частности, если t замкнут, он является элементом \mathcal{T}_n для любого n .

Заметим, что всякий (замкнутый) обыкновенный терм имеет ровно одно представление де Брауна, и что два обыкновенных терма эквивалентны с точностью до переименования связанных переменных тогда и только тогда, когда у них одинаковые представления де Брауна.

Чтобы работать с термами, содержащими свободные переменные, нам потребуется понятие *контекста именования* (naming context). Например, допустим, нам нужно представить $\lambda x. \ u \ x$ в виде безымянного терма. Мы знаем, что делать с x , но не знаем, как вести себя с u , поскольку неизвестно, как «далеко» эта переменная будет определена, и какой ей сопоставить номер. Решение состоит в том, чтобы выбрать, раз и навсегда, присвоение индексов де Брауна свободным переменным (называемое контекстом именования), и последовательно использовать это присвоение, когда требуется выбрать номер для свободной переменной. Например, предположим, что мы решили работать в следующем контексте именования:

$$\begin{aligned}\Gamma = & \quad x \mapsto 4 \\ & y \mapsto 3 \\ & z \mapsto 2 \\ & a \mapsto 1 \\ & b \mapsto 0\end{aligned}$$

Тогда x (y z) будет представлен как 4 (3 2), в то время как $\lambda w. \ u \ w$ будет представлен как $\lambda. \ 4 \ 0$, а $\lambda w. \lambda a. \ x$ — как $\lambda. \lambda. 6$.

Поскольку порядок, в котором переменные следуют в Γ , однозначно определяет их числовые индексы, мы можем кратко записать контекст в виде последовательности.

Определение 6.1.3 Допустим, x_0, \dots, x_n — имена переменных из \mathcal{V} . Контекст именования $\Gamma = x_n, x_{n-1}, \dots, x_1, x_0$ присваивает каждой x_i индекс i . Брауна i . Заметим, что самая правая переменная в контексте получает индекс 0; это соответствует тому, как мы считаем λ -связывания — справа налево, — когда преобразуем именованный терм в безымянный. Множество $\{x_n, \dots, x_0\}$ переменных, упомянутых в Γ , мы обозначаем $dom(\Gamma)$.

Упражнение 6.1.4 [★ ★ ★, \rightarrow]: Постройте альтернативную конструкцию множества n -термов в стиле определения 3.2.3, и покажите (как в утверждении 3.2.6), что ваше определение эквивалентно вышеизведенному.

Упражнение 6.1.5 [РЕКОМЕНДУЕТСЯ, ★★★]:

1. Определите функцию $removenames_\Gamma(t)$, которая принимает контекст именования Γ и обычновенный терм t (где $FV(t) \subseteq dom(\Gamma)$), и порождает соответствующий безымянный терм.
2. Определите функцию $restorenames_\Gamma(t)$, которая принимает безымянный терм t и контекст Γ , и порождает обычновенный терм. (В процессе вам придется «выдумывать» имена переменных, связанных абстракциями в t . Можете предположить, что имена в \mathcal{V} различаются попарно, и что множество имен переменных \mathcal{V} упорядочено, так что выражение «возьмем первое имя переменной, которое еще не содержится в $dom(\Gamma)$ » имеет смысл.)

Эта пара функций должна иметь свойство

$$removenames_\Gamma(restorenames_\Gamma(t)) = t$$

для любого безымянного терма t и, соответственно,

$$restorenames_\Gamma(removenames_\Gamma(t)) = t$$

с точностью до переименования связанных переменных, для любого обычновенного терма t .

Строго говоря, нельзя говорить о «некотором $t \in \mathcal{T}$ » — всегда нужно указывать, сколько свободных переменных t может иметь. Однако на практике мы обычно будем иметь в виду некоторый заранее заданный контекст именования Γ ; мы будем нескольковольно обращаться с нотацией и писать $t \in \mathcal{T}$, имея в виду $t \in \mathcal{T}_n$, где n — длина Γ .

6.2. Сдвиг и подстановка

Нашей следующей задачей будет определение операции подстановки ($[k \mapsto s]t$) на безымянных термах. Для этого потребуется вспомогательная операция, называемая «сдвигом», которая перенумеровывает индексы свободных переменных в терме.

Когда подстановка проникает внутрь λ -абстракции, к примеру, $[1 \mapsto s](\lambda.x)$ (т. е., $[x \mapsto s]\lambda.y.x$, если предположить, что 1 — индекс переменной x во внешнем контексте), контекст, в котором происходит подстановка, становится длиннее исходного на одну переменную; требуется увеличить индексы свободных переменных в s , чтобы в новом контексте они ссылались на те же переменные, что и раньше. Однако делать это нужно осторожно: нельзя просто увеличить на единицу все индексы переменных s , потому что при этом сдвинулись бы и *связанные* переменные внутри s . Например, пусть $s = 2(\lambda.0)$ (т. е., $s = z(\lambda.w.w)$, если 2 — индекс z во внешнем контексте). В этом случае нам требуется сдвинуть 2, но не 0. Функция сдвига, описанная ниже, принимает параметр «отсечки» c , управляющий тем, какие переменные сдвигаются. Исходно он равен 0 (что означает, что сдвигать нужно все переменные), и увеличивается каждый раз, когда функция сдвига пересекает границу абстракции. Таким образом, при вычислении $\uparrow_c^d(t)$ мы знаем, что терм t происходит изнутри c слоев абстракции по отношению к исходному аргументу \uparrow^d . Получается, что все идентификаторы $k < c$ внутри t связаны в исходном аргументе и сдвигу не подлежат, а идентификаторы $k \geq c$ свободны, и их нужно сдвинуть.

Определение 6.2.1 [Сдвиг]: *Сдвиг терма t на d позиций с отсечкой c , обозначаемый $\uparrow_c^d(t)$, определяется так:*

$$\begin{aligned}\uparrow_c^d(k) &= \begin{cases} k & \text{если } k < c \\ k + d & \text{если } k \geq c \end{cases} \\ \uparrow_c^d(\lambda.t_1) &= \lambda. \uparrow_{c+1}^d(t_1) \\ \uparrow_c^d(t_1 t_2) &= \uparrow_c^d(t_1) \uparrow_c^d(t_2)\end{aligned}$$

Запись $\uparrow^d(t)$ означает $\uparrow_0^d(t)$.

Упражнение 6.2.2 [★]:

1. Чему равняется $\uparrow^2(\lambda.\lambda.1(02))$?
2. Чему равняется $\uparrow^2(\lambda.01(\lambda.012))$?

Упражнение 6.2.3 [★★, →]: Покажите, что если t является n -термом и, если $d < 0$, все свободные переменные t не меньше $|d|$, то $\uparrow_c^d(t)$ является $\max(n+d, 0)$ -термом.

Теперь мы готовы определить оператор подстановки $[j \mapsto s]t$. Когда мы используем подстановку, нас обычно интересует подстановка *последней* переменной в контексте (т. е., $j = 0$), поскольку именно этот случай нам нужен, чтобы определить операцию бета-редукции. Однако для того, чтобы подставить значение переменной 0 в терме, который является лямбда-абстракцией, нужна возможность подстановки значения переменной 1 в теле этой абстракции. Таким образом, определение подстановки должно работать с произвольной переменной.

Определение 6.2.4 [Подстановка]: Подстановка терма s вместо переменной номер j в терме t , записываемая в виде $[j \mapsto s]t$, определяется следующим образом:

$$\begin{aligned}[j \mapsto s]k &= \begin{cases} s & \text{если } k = j \\ k & \text{в противном случае} \end{cases} \\ [j \mapsto s](\lambda. t_1) &= \lambda. [j+1 \mapsto \uparrow^1 s]t_1 \\ [j \mapsto s](t_1 \ t_2) &= ([j \mapsto s]t_1 [j \mapsto s]t_2) \end{aligned}$$

Упражнение 6.2.5 [★]: Переведите следующие примеры подстановок в безымянную форму в предположении глобального контекста $\Gamma = a, b$ и вычислите результаты подстановки по определению 6.2.4. Соответствуют ли ответы исходному определению подстановки на обыкновенных термах из §5.3.5?

1. $[b \mapsto a](b (\lambda x. \lambda y. b))$
2. $[b \mapsto a (\lambda z. a)](b (\lambda x. b))$
3. $[b \mapsto a](\lambda b. b a)$
4. $[b \mapsto a](\lambda a. b a)$

Упражнение 6.2.6 [★★, →]: Покажите, что если s и t — n -термы, и $j \leq n$, то $[j \mapsto s]t$ — n -терм.

Упражнение 6.2.7 [★, →]: Возьмите лист бумаги и, не глядя на определения подстановки и сдвига, сочините их заново.

Упражнение 6.2.8 [Рекомендуется, ★ ★ ★]: Определение подстановки на безымянных термах должно быть согласовано с нашим неформальным определением подстановки на обыкновенных термах. (1) Какую теорему нужно доказать для того, чтобы строго обосновать это утверждение? (2) Докажите ее.

6.3. Вычисление

Единственное, что требуется сделать, чтобы определить отношение вычисления на безымянных термах — изменить правило бета-редукции так, чтобы оно использовало нашу новую операцию подстановки (это единственное правило в старой системе, в котором упоминаются имена переменных).

Нетривиальное обстоятельство состоит в том, что редукция редекса «расходит» связанную переменную: при редукции из $((\lambda x. t_{12})v_2)$ в $[x \mapsto v_2]t_{12}$ связанная переменная x исчезает. Таким образом, переменные в результате подстановки надо перенумеровать, чтобы отразить тот факт, что x больше не является частью контекста. Например:

$$(\lambda. 1 0 2) (\lambda. 0) \rightarrow 0 (\lambda. 0) 1 \quad (\text{а не } 1 (\lambda. 0) 2)$$

Аналогично, требуется сдвинуть переменные в v_2 перед подстановкой в t_{12} , поскольку терм t_{12} определен в более крупном контексте, чем v_2 . Собирая все эти соображения вместе, получаем такое правило бета-редукции:

$$(\lambda.t_{12})\ v_2 \rightarrow \uparrow^{-1} ([0 \mapsto \uparrow^1(v_2)]t_{12}) \quad (\text{E-APP ABS})$$

Остальные правила остаются такими же, как и раньше (рис. 5.3).

Упражнение 6.3.1 [★]: Должен ли нас беспокоить тот факт, что отрицательный сдвиг в правиле может создать некорректные термы с отрицательными индексами переменных?

Упражнение 6.3.2 [★★★]: В исходной статье де Брауна описано два разных способа получения безымянного представления термов: индексы де Брауна, которые нумеруют связывающие выражения «изнутри наружу», и уровни де Брауна (*de Bruijn levels*), которые нумеруют связывающие выражения «снаружи внутрь». Например, терм $\lambda x. (\lambda y. x y) x$ представляется индексами де Брауна в виде $\lambda. (1\ 0)\ 0$, а уровнями де Брауна — в виде $\lambda. (0\ 1)\ 0$. Определите этот вариант с должной строгостью, и покажите, что представления термов с использованием индексов и уровней изоморфны (т. е., каждое из них можно однозначно восстановить, исходя из другого).

Глава 7

Реализация лямбда-исчисления на ML

В этой главе мы создадим интерпретатор для бестипового лямбда-исчисления, основываясь на интерпретаторе для арифметических выражений из главы 4, с использованием подхода к связыванию переменных и к подстановке, описанному в главе 6.

Выполняемый вычислитель бестиповых лямбда-термов может быть получен путем простого перевода на OCaml определений из предыдущих глав. Как и в главе 4, мы демонстрируем только базовые алгоритмы, игнорируя вопросы лексического и синтаксического анализа, ввода-вывода и тому подобные детали.

7.1. Термы и контексты

Тип данных, представляющий абстрактные синтаксические деревья для термов, можно получить путем прямого перевода определения 6.1.2:

```
type term =
  TmVar of int
  | TmAbs of term
  | TmApp of term * term
```

Представлением переменной является число — ее индекс де Брауна. Представление абстракции содержит только терм для тела абстракции. Применение содержит два терма, один из которых применяется к другому.

Однако определение, которое реально используется в нашем интерпретаторе, содержит несколько больше информации. Во-первых, как и раньше, каждый терм полезно снабдить элементом типа `info`, в котором записано, из какой позиции в файле терм был считан, чтобы программы распечатки ошибок

В основном в этой главе обсуждается чистое бестиповое лямбда-исчисление (рис. 5.3). Соответствующая реализация называется `untyped`. Интерпретатор `fulluntyped` содержит расширения, например, поддержку чисел и булевских значений.

могли указать пользователю (может быть, даже автоматически, с помощью текстового редактора) точное место, в котором произошла ошибка.

```
type term =
  TmVar of info * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

Во-вторых, для отладки полезно хранить в каждой вершине-переменной дополнительное число для проверки целостности. Мы будем использовать такое соглашение: в этом втором числовом поле всегда содержится *полная длина* контекста, в котором встретилась эта переменная.

```
type term =
  TmVar of info * int * int
  | TmAbs of info * term
  | TmApp of info * term * term
```

Каждый раз при распечатке переменной мы будем проверять это число на совпадение с реальной длиной контекста, в котором мы находимся; несовпадение означает, что мы где-то забыли добавить операцию сдвига.

Последнее усовершенствование тоже относится к распечатке. Несмотря на то, что внутри термы представляются через индексы де Брауна, пользователю, они, очевидно, в таком виде показываться не должны: при чтении следует преобразовывать термы из обычного представления в безымянное, а при распечатке — преобразовывать их обратно в обычную форму. В этом нет ничего сложного, однако проделывать это совсем наивным образом (скажем, порождая для всех переменных совершенно новые имена) не стоит, поскольку тогда имена связанных переменных в печатаемых термах никак не будут связаны с именами из исходной программы. Это затруднение можно преодолеть, если хранить при каждой абстракции строку-подсказку с именем связанной переменной.

```
type term =
  TmVar of info * int * int
  | TmAbs of info * string * term
  | TmApp of info * term * term
```

Основные операции с термами (в частности, подстановка) ничего особенного с этими строками не делают: они просто переносятся в результат в исходной форме без всякой заботы о совпадении имен, захвате переменных, и т. д. Когда программе распечатки требуется породить новое имя для связанной переменной, она сначала пытается использовать подсказку; если окажется, что это имя уже используется в текущем контексте, процедура распечатки пытается породить похожее имя, добавляя к имени штрихи ('), пока в конце концов не найдется имя, еще не используемое в данном контексте. Благодаря этому правилу напечатанный терм будет всегда очень похож на то, чего ожидает пользователь, с точностью до нескольких штрихов.

Сама процедура печати выглядит так:

```
let rec printtm ctx t = match t with
  TmAbs(fi,x,t1) →
```

```

let (ctx',x') = pickfreshname ctx x in
  pr "(lambda"; pr x'; pr ".u"; printtm ctx' t1; pr ")"
| TmApp(fi, t1, t2) →
  pr "("; printtm ctx t1; pr "u"; printtm ctx t2; pr ")"
| TmVar(fi,x,n) →
  if ctxlength ctx = n then
    pr (index2name fi ctx x)
  else
    pr "[badindex]"

```

В ней используется тип данных `context`,

```
type context = (string * binding) list
```

который представляет собой простой список строк и соответствующих им связываний. Пока что связывания совершенно тривиальны

```
type binding = NameBind
```

и не несут никакой дополнительной информации. Однако позднее (в главе 10) мы введем в определение типа `binding` другие варианты, которые будут отслеживать сведения о типе, связанном с переменной, и другую подобную информацию.

Процедура распечатки также использует нескольких низкоуровневых функций: `pr` выдает строку в стандартный поток вывода; `ctxlength` возвращает длину контекста; `index2name` находит строковое имя переменной по ее индексу. Самая интересная из этих функций — `pickfreshname`, которая принимает контекст `ctx` и имя-подсказку `x`, находит имя `x'`, похожее на `x` и не встречающееся в `ctx`, добавляет `x'` к контексту `ctx`, получая при этом новый контекст `ctx'`, и возвращает пару, состоящую из `x'` и `ctx'`.

Реальная процедура печати в интерпретаторе `untyped`, имеющаяся на веб-сайте книги, выглядит несколько сложнее, поскольку принимает во внимание еще два обстоятельства. Во-первых, она по возможности избегает печатать скобки, следуя соглашению о том, что применение право-ассоциативно, а тела абстракций простираются насколько возможно вправо. Во-вторых, она порождает команды форматирования для OCaml-библиотеки `Format` — низкоуровневого модуля *красивой печати* (pretty printing), который принимает решения о переводах строк и отступах.

7.2. Сдвиг и подстановка

Определение сдвига (6.2.1) переводится на OCaml практически посимвольно.

```

let termShift d t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → if x>=c then TmVar(fi,x+d,n+d)
                     else TmVar(fi,x,n+d)
    | TmAbs(fi,x,t1) → TmAbs(fi, x, walk (c+1) t1)
    | TmApp(fi,t1,t2) → TmApp(fi, walk c t1, walk c t2)
  in walk 0 t

```

Внутренний сдвиг $\uparrow_c^d(t)$ здесь представлен вызовом внутренней функции `walk` с `t`. Поскольку `d` не меняется, нет нужды передавать ее в каждый вызов `walk`: когда это требуется в варианте с переменной внутри `walk`, мы просто используем внешнее связывание `d`. Сдвиг верхнего уровня $\uparrow^d(t)$ представляется выражением `termShift d t`. (Заметим, что сама функция `termShift` не помечена как рекурсивная, поскольку ее единственная задача — один раз вызывать `walk`.)

Аналогично, функция подстановки получается почти напрямую из определения 6.2.4:

```
let termSubst j s t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → if x=j+c then termShift c s else TmVar(fi,x,n)
  | TmAbs(fi,x,t1) → TmAbs(fi,x,walk(c+1)t1)
  | TmApp(fi,t1,t2) → TmApp(fi,walk c t1,walk c t2)
  in walk 0 t
```

Подстановка $[j \mapsto s]t$ терма `s` вместо переменной `j` в терме `t` записывается здесь в виде `termSubst j s t`. Единственное отличие от исходного определения подстановки состоит в том, что весь сдвиг `s` производится сразу, в ветви `TmVar`, вместо того, чтобы сдвигать `s` на единицу при каждом проходе через связывание. При этом получается, что аргумент `j` во всех вызовах `walk` один и тот же, и во внутреннем определении его можно опустить.

Читатель может заметить, что определения `termShift` и `termSubst` весьма похожи, и отличаются только действием, производимым, когда процесс достигает переменной. Интерпретатор `untyped`, имеющийся на веб-сайте книги, использует это сходство и выражает как сдвиг, так и подстановку в качестве частных случаев более общей функции `tmmap`. Принимая терм `t` и функцию `onvar`, `tmmap onvar t` выдает терм той же формы, что `t`, в котором каждая переменная заменена на результат вызова `onvar` от этой переменной. В более крупных исчислениях такой прием избавляет нас от довольно большого количества повторений; подробности можно найти в §25.2.

Единственное место в операционной семантике лямбда-исчисления, в котором используется подстановка — это правило бета-редукции. Как мы уже замечали, это правило на самом деле производит несколько операций: терм, подставляемый вместо связанной переменной, сначала сдвигается на единицу вверх, а затем результат сдвигается на единицу вниз, чтобы отразить исчезновение использованной связанной переменной. Следующее определение описывает эту последовательность действий:

```
let termSubstTop s t =
  termShift (-1) (termSubst 0 (termShift 1 s) t)
```

7.3. Вычисление

Как и в главе 4, функция вычисления зависит от вспомогательного предиката `isval`:

```
let rec isval ctx t = match t with
```

```
TmAbs(_, _, _) → true
| _ → false
```

Функция одношагового вычисления прямо кодирует правила вычисления, но при этом, помимо терма, принимает дополнительным аргументом контекст `ctx`. В нашей нынешней функции `eval1` этот контекст не используется, но некоторые более сложные вычислители в последующих главах будут к нему обращаться.

```
let rec eval1 ctx t = match t with
  TmApp(fi, TmAbs(_, x, t12), v2) when isval ctx v2 →
    termSubstTop v2 t12
  | TmApp(fi, v1, t2) when isval ctx v1 →
    let t2' = eval1 ctx t2 in
      TmApp(fi, v1, t2')
  | TmApp(fi, t1, t2) →
    let t1' = eval1 ctx t1 in
      TmApp(fi, t1', t2)
  | _ →
    raise NoRuleApplies
```

Функция многошагового вычисления такая же, как раньше, за исключением аргумента `ctx`:

```
let rec eval ctx t =
  try let t' = eval1 ctx t
    in eval ctx t'
  with NoRuleApplies → t
```

Упражнение 7.3.1 [РЕКОМЕНДУЕТСЯ, ★★★ →]: Измените реализацию так, чтобы она использовала стиль вычисления с «большим шагом», введенный в упражнении 3.5.17.

7.4. Дополнительные замечания

Реализация подстановок, представленная в этой главе, хотя и достаточна для целей нашей книги, но далека от идеала. В частности, правило бета-редукции в нашем интерпретаторе «энергично» подставляет значение аргумента вместо связанной переменной в теле процедуры. Интерпретаторы (и компиляторы) функциональных языков, которые оптимизируют скорость, а не простоту чтения кода, используют другую стратегию: вместо того, чтобы производить подстановку, они просто записывают информацию о связи между именем переменной и значением аргумента во вспомогательной структуре данных, называемой *окружением* (environment), и эта структура передается вместе с вычисляемым термом. При обращении к переменной мы ищем ее значение в текущем окружении. Такую стратегию можно смоделировать, если рассматривать окружение как своего рода *явную подстановку* (explicit substitution) — т. е., перевести механизм подстановки из метаязыка в объектный язык, сделать его частью *синтаксиса* термов, с которыми работает вычислитель, а не внешней операцией на термах. Явные подстановки появились

в исследовании Абади, Карделли, Куриена и Леви (Abadi, Cardelli, Curien, and Lévy, 1991a) и стали с тех пор активной областью исследований.

Если вы что-то реализовали в
программе, это еще не значит,
что вы это поняли.

Брайан Кантвелл Смит

Часть II

Простые типы

Глава 8

Типизированные арифметические выражения

В главе 3 мы с помощью простого языка булевых и арифметических выражений продемонстрировали основные инструменты для точного описания синтаксиса и вычислений. Теперь мы вернемся к этому языку и дополним его статическими типами. Как и в главе 3, сама по себе система типов совершенно тривиальна, однако она поможет нам познакомиться с понятиями, которые затем будут использоваться на протяжении всей книги.

8.1. Типы

Напомним синтаксис арифметических выражений:

$t ::=$	<i>термы:</i>
true	константа «истина»
false	константа «ложь»
if t then t else t	условное выражение
0	константа «ноль»
succ t	следующее число
pred t	предыдущее число
iszero t	проверка на ноль

В главе 3 мы видели, что при вычислении терма может либо получиться значение:

Система, изучаемая в этой главе — типизированное исчисление булевых значений и чисел (рис. 8.2). Соответствующая реализация на OCaml называется `tyarith`.

v	::=	...	значения:
		true	константа «истина»
		false	константа «ложь»
		nv	числовое значение
nv	::=	...	числовые значения:
		0	нулевое значение
		succ nv	значение-последователь

либо вычисление может на каком-то шаге зайти в *тупик*, наткнувшись на терм *вроде pred false*, к которому не применимы никакие правила.

Тупиковые термы соответствуют бессмысленным или ошибочным программам. Поэтому, мы хотели бы иметь возможность удостовериться в том, что вычисление данного терма наверняка *не зайдет* в тупик, не производя само вычисление. Для этого мы должны уметь отличать термы, значение которых будет числовым (поскольку только эти термы могут использоваться в качестве аргументов для *pred*, *succ* и *iszzero*) от тех, значение которых будет булевским (только такие термы могут служить условием в условном выражении). Для такой классификации термов мы вводим два *типа*, *Nat* и *Bool*. На протяжении всей книги метапеременные *S*, *T*, *U* и т. п. будут обозначать типы.

Утверждение «терм *t* имеет тип *T*» (или «*t* принадлежит типу *T*», или «*t* является элементом *T*») означает, что *t* «очевидным образом» дает при вычислении значение нужного вида — при этом под словами «очевидным образом» мы подразумеваем, что проверить это можно *статически* (statically), не производя само вычисление *t*. Например, терм *if true then false else true* имеет тип *Bool*, а *pred (succ (pred (succ 0)))* имеет тип *Nat*. Однако наш анализ типов будет *консервативным* (conservative), то есть, будет обращаться только к статической информации. Это означает, что мы не сможем определить, имеют ли термы *вроде if (iszzero 0) then 0 else false*, или даже *if true then 0 else false* какой-либо тип, даже при том, что их вычисление на самом деле в тупик не заходит.

8.2. Отношение типизации

Отношение типизации для арифметических выражений, записываемое в виде¹ «*t : T*», определяется набором правил вывода, присваивающих термам типы. Эти правила перечислены на рис. 8.1 и 8.2. Как и в главе 3, мы помещаем правила для булевых значений и для чисел на два разных рисунка, поскольку впоследствии нам иногда будет нужно ссылаться на них по отдельности.

Правила *T-TRUE* и *T-FALSE* на рис. 8.1 присваивают булевским константам *true* и *false* тип *Bool*. Правило *T-IF* присваивает тип условному выражению на основании типов его подвыражений: условие *t₁* должно при вычислении давать булевское значение, а *t₂* и *t₃* должны давать значения *одного и того же* типа. Два упоминания метапеременной *T* выражают ограничение, согласно которому тип результата вычисления *if* совпадает с типом ветвей *then* и

¹Часто вместо : используется символ \in .

В (типовизированное)	<i>Новые синтаксические формы</i> $T ::= \begin{array}{l} \text{типы:} \\ \text{Bool} \end{array}$ <i>тип булевских значений</i>	<i>Расширяет В (3.1)</i> $t : T$
		<i>Новые правила типизации</i>
		$\text{true} : \text{Bool}$ (T-TRUE) $\text{false} : \text{Bool}$ (T-FALSE)
		$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$ (T-IF)

Рис. 8.1. Правила типизации для булевых значений (В).

`else`, причем этот тип может быть любым (либо `Nat`, либо `Bool`, а также, когда мы доберемся до исчислений с более интересными наборами типов, любой другой тип).

Правила для чисел на рис. 8.2 имеют аналогичный вид. `T-ZERO` присваивает константе 0 тип `Nat`. `T-SUCC` присваивает выражению `succ t1` тип `Nat` при условии, что `t1` имеет тип `Nat`. Аналогично, `T-PRED` и `T-IsZERO` говорят, что `pred` дает в результате `Nat`, если его аргумент имеет тип `Nat`, а `iszero` дает результат типа `Bool` при аргументе типа `Nat`.

Определение 8.2.1 С формальной точки зрения, отношение типизации (*typing relation*) для арифметических выражений — это наименьшее бинарное отношение между термами и типами, удовлетворяющее всем правилам с рис. 8.1 и 8.2. Терм `t` является типизируемым (*typable*) (или корректно типизированным, *well-typed*), если существует тип `T` такой, что `t : T`.

В рассуждениях об отношениях типизации мы часто будем делать утверждения вроде «Если терм вида `succ t1` имеет какой-нибудь тип, то это должен быть тип `Nat`». Нижеследующая лемма дает нам набор утверждений такого вида. Каждое из них немедленно следует из формы соответствующего правила типизации.

Лемма 8.2.2 [Инверсия отношения типизации]:

1. Если `true : R`, то `R = Bool`.
2. Если `false : R`, то `R = Bool`.
3. Если `if t1 then t2 else t3 : R`, то `t1 : Bool`, `t2 : R` и `t3 : R`.
4. Если `0 : R`, то `R = Nat`.

BN (типованием)

Расширяет NB (3.2) и 8.1

Новые синтаксические формы

$$\begin{array}{l} T ::= \dots \\ Nat \text{ тип натуральных чисел} \end{array}$$

Новые правила типизации

$$\begin{array}{l} t : T \\ 0 : Nat \end{array}$$
 $t : T$

(T-ZERO)

$$\frac{t_1 : Nat}{succ\ t_1 : Nat}$$

(T-SUCC)

$$\frac{t_1 : Nat}{pred\ t_1 : Nat}$$

(T-PRED)

$$\frac{t_1 : Nat}{iszero\ t_1 : Bool}$$

(T-ISZERO)

Рис. 8.2. Правила типизации для чисел (NB).

5. Если $succ\ t_1 : R$, то $R = Nat$ и $t_1 : Nat$.
6. Если $pred\ t_1 : R$, то $R = Nat$ и $t_1 : Nat$.
7. Если $iszero\ t_1 : R$, то $R = Bool$ и $t_1 : Nat$.

Доказательство: Непосредственно следует из определения отношения типизации.

Лемму об инверсии типизации часто называют *леммой о порождении* (generation lemma) для отношения типизации, поскольку она позволяет, имея верное утверждение о типе терма, понять, как можно породить доказательство этого утверждения. Лемма об инверсии непосредственно приводит к рекурсивному алгоритму, вычисляющему типы термов, поскольку для терма каждой синтаксической формы она показывает, как вычислить его тип (если таковой имеется), исходя из типов его подтермов. Мы подробно рассмотрим это утверждение в главе 9.

Упражнение 8.2.3 [★ →]: Докажите, что все подтермы типизируемого терма также типизируются.

В §3.5 мы познакомились с понятием деревьев вывода для вычислений. Аналогичным образом, *дерево вывода типов* (typing derivation) представляет собой дерево, состоящее из экземпляров правил типизации. Каждой паре (t, T) в отношении типизации соответствует дерево вывода типов с заключением $t : T$. Вот, например, дерево вывода для утверждения типизации «*if iszero 0 then 0 else pred 0 : Nat*»:

$$\frac{\overline{\quad} \text{T-ZERO}}{0 : Nat} \quad
 \frac{\overline{\quad} \text{T-ZERO} \quad \overline{\quad} \text{T-IsZERO}}{iszero\ 0 : Bool} \quad
 \frac{\overline{\quad} \text{T-ZERO}}{0 : Nat} \quad
 \frac{\overline{\quad} \text{T-ZERO} \quad \overline{\quad} \text{T-PRED}}{pred\ 0 : Nat} \quad
 \frac{}{\text{if iszero 0 then 0 else pred 0 : Nat}} \text{ T-IF}$$

Иначе говоря, *утверждения* (statements) — это формальные высказывания о типах в программах, *правила типизации* (typing rules) — это импликации между утверждениями, а *деревья вывода* (typing derivations) — доказательства, построенные с помощью правил типизации.

Теорема 8.2.4 [Единственность типов]: *Всякий терм t имеет не более одного типа. То есть, если t типизируем, то у него есть единственный тип. Более того, есть только один способ вывести этот тип с помощью правил рис. 8.1 и 8.2.*

Доказательство: Прямолинейная структурная индукция по t . В каждом варианте используется соответствующее утверждение леммы инверсии и предположение индукции.

В простой системе типов, с которой мы работаем в этой главе, у каждого терма только один тип (если тип у него вообще есть), и для подтверждения этого всегда есть всего одно дерево вывода. Позже, скажем в главе 15, когда мы доберемся до систем с подтиปами, оба этих свойства будут ослаблены: один терм может иметь несколько типов, и в общем случае может существовать несколько способов доказательства утверждения о том, что данный терм имеет данный тип.

Свойства отношения типизации часто будут доказываться при помощи индукции на деревьях вывода типов, так же, как свойства отношения вычисления обычно доказываются индукцией на деревьях вывода вычисления. Начиная со следующего раздела, мы встретим много примеров индукции на деревьях вывода типов.

8.3. Безопасность = продвижение + сохранение

Основное свойство нашей (и любой другой) системы типов — *безопасность* (также называемая *корректностью* (soundness)): правильно типизированные термы «никогда не ломаются». Мы уже договорились о формализации понятия «поломки» терма: это значит, что терм оказался в «тупиковом состоянии» (Определение 3.5.15), в котором терм не является окончательным значением, но правила вычисления ничего не говорят о том, что с ним делать дальше. Следовательно, мы хотим быть уверены в том, что правильно типизированные термы никогда не оказываются в тупике. Мы доказываем это в два шага, которые традиционно называют теоремами *продвижения* (progress) и *сохранения* (preservation).²

Продвижение: Правильно типизированный терм не может быть тупиковым (либо это значение, либо он может проделать следующий шаг в соответствии с правилами вычисления).

²Лозунг «безопасность — это продвижение плюс сохранение» (с использованием леммы о канонических формах) был сформулирован Харпером; вариант того же лозунга был предложен Райтом и Феллрайсеном (Wright and Felleisen, 1994).

Сохранение: Если правильно типизированный терм проделывает шаг вычисления, получающийся терм также правильно типизирован.³

Вместе эти два свойства гарантируют, что правильно типизированный терм никогда не попадет в тупик в процессе вычисления.

Для доказательства теоремы о продвижении удобно записать несколько утверждений о возможном виде *канонических форм* (canonical forms) типов *Bool* и *Nat* (т. е., правильно типизированных значениях этих типов).

Лемма 8.3.1 [КАНОНИЧЕСКИЕ ФОРМЫ]:

1. Если v — значение типа *Bool*, то v равно либо *true*, либо *false*.
2. Если v — значение типа *Nat*, то v является числовым значением согласно грамматике, представленной на рис. 3.2.

Доказательство: Часть (1): согласно грамматике, представленной на рис. 3.1 и 3.2, значения в этом языке могут иметь четыре формы: *true*, *false*, *0* и *succ n*, где *n* — числовое значение. Первые два варианта немедленно дают нам требуемый результат. Два оставшихся не могут возникнуть, поскольку v , по предположению, имеет тип *Bool*, а по пунктам 4 и 5 леммы об инверсии получается, что *0* и *succ n* могут иметь только тип *Nat*, а не *Bool*. Часть (2) доказывается аналогично.

Теорема 8.3.2 [ПРОДВИЖЕНИЕ]: Допустим, что t — корректно типизированный терм (то есть, $t : T$ для некоторого типа T). Тогда либо t является значением, либо существует некоторый t' , такой, что $t \rightarrow t'$.

Доказательство: Индукция по дереву вывода $t : T$. Варианты *T-TRUE*, *T-FALSE* и *T-ZERO* дают требуемый результат немедленно, так как в этих случаях t — значение. В остальных случаях мы рассуждаем так:

Вариант T-IF: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
 $t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

Согласно предположению индукции, либо t_1 является значением, либо существует терм t'_1 , такой что $t_1 \rightarrow t'_1$. Если t_1 — значение, то, как следует из леммы о канонических формах, t_1 является либо *true*, либо *false*, а следовательно, к t можно применить либо *E-IFTRUE*, либо *E-IFFALSE*. С другой стороны, если $t_1 \rightarrow t'_1$, то, по правилу *E-IF*, $t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$.

Вариант T-SUCC: $t = \text{succ } t_1 \quad t_1 : \text{Nat}$

Согласно предположению индукции, либо t_1 является значением, либо существует какой-то t'_1 , такой, что $t_1 \rightarrow t'_1$. Если t_1 является значением, то, по лемме о канонических формах, это должно быть числовое значение, а тогда t — тоже значение. С другой стороны, если $t_1 \rightarrow t'_1$, то, по правилу *E-SUCC*, $\text{succ } t_1 \rightarrow \text{succ } t'_1$.

³ В большинстве рассматриваемых нами систем вычисление сохраняет не только типизируемость, но и сам тип термов. Однако в некоторых системах тип при вычислении может меняться. Например, в системах с подтипами (глава 15) типы в процессе вычисления могут становиться меньше (более информативными).

Вариант T-PRED: $t = \text{pred } t_1 \quad t_1 : \text{Nat}$

Согласно предположению индукции, либо t_1 является значением, либо существует какой-то t'_1 , такой, что $t_1 \rightarrow t'_1$. Если t_1 является значением, то, по лемме о канонических формах, это должно быть числовое значение, либо 0, либо $\text{succ } n_{v_1}$ для некоторого n_{v_1} , и тогда к t применимо одно из правил E-PREDZERO или E-PREDSUCC. Если, с другой стороны, $t_1 \rightarrow t'_1$, то, по правилу E-PRED, $\text{pred } t_1 \rightarrow \text{pred } t'_1$.

Вариант T-IsZERO: $t = \text{iszero } t_1 \quad t_1 : \text{Nat}$

Доказывается аналогичным образом.

Доказательство того, что типы сохраняются при вычислении, в этой системе также не представляет труда.

Теорема 8.3.3 [СОХРАНЕНИЕ]: *Если $t : T$ и $t \rightarrow t'$, то $t' : T$.*

Доказательство: Индукция по дереву вывода $t : T$. На каждом шаге индукции мы предполагаем, что требуемое свойство выполняется для всех поддеревьев (*т. е., если $s : S$ и $s \rightarrow s'$, то $s' : S$, когда $s : S$ доказано в поддереве текущего вывода*), и рассматриваем варианты последнего правила в выводе типа (мы демонстрируем лишь некоторые из вариантов; остальные устроены аналогично).

Вариант T-TRUE: $t = \text{true} \quad T = \text{Bool}$

Если последнее правило вывода — T-TRUE, то из формы этого правила нам известно, что t — константа true , а T — тип Bool . Но в таком случае, t является значением, поэтому невозможно, чтобы для какого-либо t' имелось $t \rightarrow t'$, и требования теоремы просто не могут нарушаться.

Вариант T-IF: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \quad t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

Если последнее правило в выводе типа — T-IF, то из формы этого правила нам известно, что t имеет вид $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$ для некоторых t_1 , t_2 и t_3 . Кроме того, должны существовать поддеревья вывода с заключениями $t_1 : \text{Bool}$, $t_2 : T$ и $t_3 : T$. При взгляде на правила вычисления, в которых левая сторона представляет собой условное выражение, мы видим три правила, которые могут привести к заключению $t \rightarrow t'$: E-IFTURE, E-IFFALSE и E-IF. Рассмотрим их по отдельности (кроме E-IFFALSE, которое аналогично E-IFTURE):

Подвариант E-IFTURE: $t_1 = \text{true} \quad t' = t_2$

Если $t \rightarrow t'$ выводится по правилу E-IFTURE, то из формы этого правила мы видим, что t_1 должен равняться true , а результат t' должен равняться второму подвыражению t_2 . Следовательно, требуемое утверждение доказано, так как нам известно (из предположений для варианта T-IF), что $t_2 : T$, чего мы и добиваемся.

Подвариант E-IF: $t_1 \rightarrow t'_1 \quad t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$

Из предположений для варианта T-IF нам известно, что в исходном дереве вывода есть поддерево с заключением $t_1 : \text{Bool}$. К

этому поддереву мы можем применить предположение индукции и получить $t'_1 : \text{Bool}$. В сочетании с известными нам (из предположений для варианта T-IF) утверждениями, что $t_2 : T$ и $t_3 : T$, это дает нам право применить правило T-IF и заключить, что *if* t'_1 *then* t_2 *else* $t_3 : T$, или, что то же самое, $t' : T$.

Вариант T-ZERO: $t = 0 \quad T = \text{Nat}$

Невозможно (по тем же причинам, что и T-TRUE).

Вариант T-SUCC: $t = \text{succ } t_1 \quad T = \text{Nat} \quad t_1 : \text{Nat}$

Рассматривая правила вычисления на рис. 3.2, мы видим, что есть только одно правило, E-SUCC, пригодное для вывода $t \rightarrow t'$. Из формы этого правила ясно, что $t_1 \rightarrow t'_1$. Поскольку мы знаем, что $t_1 : \text{Nat}$, мы можем по предположению индукции заключить, что $t'_1 : \text{Nat}$, а отсюда, по правилу T-SUCC, $\text{succ } t'_1 : \text{Nat}$, т. е., $t' : T$.

Упражнение 8.3.4 [★★ →]: Перестройте доказательство так, чтобы индукция велась не по деревьям вывода типизации, а по деревьям вывода для отношения вычисления.

Теорему о сохранении часто называют теоремой о *редукции субъекта* (subject reduction) (или о *вычислении субъекта*, subject evaluation). Интуитивное представление здесь заключается в том, что утверждение о типизации $t : T$ можно интерпретировать как предложение « t имеет тип T ». Терм t является подлежащим (субъектом) этого предложения, и свойство редукции субъекта говорит в таком случае, что истинность предложения сохраняется при редукции субъекта.

В отличие от единственности типов, которая присутствует в одних системах и отсутствует в других, продвижение и сохранение станутся важнейшими требованиями для всех систем типов, которые нам предстоит рассмотреть.⁴

Упражнение 8.3.5 [★]: Правило вычисления E-PREDZERO (рис. 3.2) не совсем согласуется с интуитивным представлением: кажется, что было бы естественней считать, что предшественник нуля не определен, а не определять его как ноль. Можно ли добиться этого, просто убрав правило E-PREDZERO из определения одношагового вычисления?

Упражнение 8.3.6 [РЕКОМЕНДУЕТСЯ, ★★]: Рассмотрев свойство редукции субъекта, можно задуматься о том, насколько выполняется обратное свойство — расширение субъекта. Всегда ли, если $t \rightarrow t'$ и $t' : T$, верно, что $t : T$? Если да, докажите это; если нет, приведите контрпример.

⁴Существуют языки, в которых эти свойства не выполняются, притом, что их все же можно считать безопасными с точки зрения типов. Например, при формализации операционной семантики Java в стиле с малым шагом (Flatt, Krishnamurthi, and Felleisen, 1998a; Igarashi, Pierce, and Wadler, 1999), сохранение типов в описанной здесь форме отсутствует (подробности см. в главе 19). Однако это обстоятельство следует рассматривать как особенность конкретного подхода к формализации, а не как дефект самого языка, поскольку она исчезает, например, при переходе к семантике с большим шагом.

Упражнение 8.3.7 [РЕКОМЕНДУЕТСЯ, ★★]: *Допустим, что наше отношение вычисления определено в стиле с большим шагом, как в упражнении 3.5.17. Как в таком случае нужно формализовать интуитивное свойство безопасности типов?*

Упражнение 8.3.8 [РЕКОМЕНДУЕТСЯ, ★★]: *Допустим, что наше отношение вычисления расширено так, чтобы бессмысленные термы сводились к явному состоянию `wrong`, как в упражнении 3.5.16. Как в этом случае формализовать безопасность типов?*

Путь от бестиповых систем к типизированным был пройден многократно, в различных областях знания и, во многом, по одним и тем же причинам.

Лука Карделли и Питер Вегнер
(Cardelli and Wegner, 1985)

Глава 9

Простое типизированное лямбда-исчисление

В этой главе вводится самый простой представитель семейства типизированных исчислений, которые мы будем изучать в оставшейся части книги: простое типизированное лямбда-исчисление Чёрча (Church, 1940) и Карри (Curry and Feys, 1958).

9.1. Типы функций

В главе 8 мы определили простую статическую систему типов для арифметических выражений с двумя типами: типом `Bool`, который относится к термам, вычисление которых дает булевские значения, и типом `Nat`, который относится к термам, при вычислении которых получаются числа. К «плохо типизированным» термам, не подпадающим ни под один из этих типов, относятся все термы, которые при вычислении попадают в тупиковое состояние (например, `if 0 then 1 else 2`), а также некоторые термы, которые на самом деле при вычислении ведут себя корректно, но для которых наша статическая классификация оказывается слишком жёсткой (скажем, `if true then 0 else false`).

Предположим, что нам нужно построить подобную систему типов для языка, в котором булевские значения (для краткости в этой главе мы забудем про числа) сочетаются с примитивами чистого лямбда-исчисления. А именно, мы хотим ввести правила типизации для переменных, абстракций и применений, которые а) поддерживают типовую безопасность — т. е. удовлетворяют теоремам о продвижении (8.3.2) и сохранении (8.3.3), и при этом б) не слишком консервативны, т. е. должны уметь присваивать типы большинству программ, которые нам на самом деле хотелось бы написать.

Разумеется, поскольку чистое лямбда-исчисление полно по Тьюрингу, мы не можем даже надеяться, что нам удастся провести *точный* анализ типов для

В этой главе изучается простое типизированное лямбда-исчисление (рис. 9.1) с булевскими значениями (рис. 8.1). Соответствующая реализация на OCaml называется `fullsimple`.

всех примитивов. Например, нет никакого надежного способа понять, возвращает ли программа

```
if <длгое и сложное вычисление> then true else ( $\lambda x. x$ )
```

булевское значение или функцию, не запустив сначала это долгое и сложное вычисление и не посмотрев, выдало ли оно истинное или ложное значение. В общем случае, долгое и сложное вычисление может вообще не завершиться, и тогда всякий вычислитель, который попытается точно предсказать его исход, тоже никогда не выдаст результата.

Ясно, что если мы хотим расширить систему типов с булевскими значениями и добавить к ней функции, то нам нужно ввести тип, относящийся к термам, вычисление которых дает в результате функцию. В качестве первого приближения назовем этот тип \rightarrow («стрелка»). Если мы введем правило типизации

$$\lambda x. t : \rightarrow$$

дающее всякой λ -абстракции тип \rightarrow , то мы можем классифицировать простые термы, например $\lambda x. x$, а также сложные, например `if true then ($\lambda x. true$) else ($\lambda x. \lambda y. y$)`, как возвращающие при вычислении функцию.

Однако такой грубый анализ явно чересчур консервативен: например, функции $\lambda x. true$ и $\lambda x. \lambda y. y$ попадают в одну группу и получают один и тот же тип. При этом игнорируется тот факт, что применение одной функции к аргументу дает булевское значение, а применение другой снова дает функцию. В общем случае, чтобы придать осмысленный тип результату применения, нужно не просто знать, что его левая часть — функция; нужно знать, какой именно тип эта функция возвращает. Более того, чтобы быть уверенными, что функция поведет себя правильно при вызове, нужно выяснить, аргументы какого типа она ожидает. Для сбора этой информации мы заменим простой тип \rightarrow бесконечным семейством типов вида $T_1 \rightarrow T_2$, каждый из которых описывает функции, которые принимают аргументы типа T_1 и возвращают результаты типа T_2 .

Определение 9.1.1 Множество простых типов (*simple types*) на основе типа `Bool` порождается следующей грамматикой:

$$\begin{aligned} T ::= & \quad \text{типы:} \\ & \text{Bool} \quad \text{тип булевских значений} \\ & T \rightarrow T \quad \text{тип функций} \end{aligned}$$

Конструктор типов (*type constructor*) \rightarrow правово-ассоциативен, то есть выражение $T_1 \rightarrow T_2 \rightarrow T_3$ обозначает $T_1 \rightarrow (T_2 \rightarrow T_3)$.

Например, $\text{Bool} \rightarrow \text{Bool}$ — это тип функций, переводящих булевские аргументы в булевские результаты. $(\text{Bool} \rightarrow \text{Bool}) \rightarrow (\text{Bool} \rightarrow \text{Bool})$, или, что то же самое, $(\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \text{Bool}$ — это тип функций, принимающих функции, переводящие булевские значения в булевские значения, и возвращающих такие же функции.

9.2. Отношение типизации

Чтобы определить тип абстракции вида $\lambda x. t$, нужно понять, что случится, когда эта абстракция будет применена к какому-либо аргументу. Немедленно возникает вопрос: откуда мы знаем, аргументов какого типа следует ожидать? На этот вопрос возможны два ответа: либо мы просто помечаем λ -абстракцию типом ожидаемого аргумента, либо мы анализируем ее тело, смотрим, как используется в нем аргумент, и пытаемся определить, какого типа он должен быть. Сейчас мы выбираем первый вариант. Вместо выражения $\lambda x. t$ мы будем писать $\lambda x:T_1. t_2$. Аннотация при связанной переменной говорит нам, что аргумент имеет тип T_1 .

Языки, в которых для помощи процедуре проверки типов используются аннотации на термах, называются *явно типизированными* (explicitly typed). Языки, где программа проверки типов пытается *вывести* (infer) или *реконструировать* (reconstruct) эту информацию, называются *неявно типизированными* (implicitly typed). (В литературе по λ -исчислению используется также термин *системы присвоения типов*, type-assignment systems.) По большей части в этой книге рассматриваются явно типизированные языки; неявной типизации посвящена глава 22.

Когда известен тип аргумента, ясно, что тип результата функции совпадает с типом тела t_2 , полученным исходя из предположения, что вхождения x внутри t_2 обозначают термы типа T_1 . Эта интуитивная идея выражается следующим правилом типизации:

$$\frac{x:T_1 \vdash t_2:T_2}{\vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-Abs})$$

Поскольку термы могут содержать вложенные λ -абстракции, в общем случае нам придется говорить о нескольких подобных предположениях. Таким образом, отношение типизации из двухместного, $t : T$, становится трехместным, $\Gamma \vdash t : T$, где Γ — набор предположений о типах свободных переменных в t .

Формально, *контекст типизации* (typing context) (также называемый *окружение типизации*, typing environment) Γ представляет собой последовательность переменных и их типов, а оператор «запятая» расширяет Γ , добавляя к нему справа новое связывание. Пустой контекст иногда изображается символом \emptyset , однако чаще всего мы просто опускаем его и пишем $\vdash t : T$, что означает: «Замкнутый терм t имеет тип T , исходя из пустого множества предположений».

Чтобы избежать конфликтов между новым связыванием и связываниями, уже присутствующими в Γ , мы требуем, чтобы имя x отличалось от переменных, связанных в Γ . Поскольку у нас действует соглашение, что переменные, связанные λ -абстракциями, разрешается переименовывать, это требование всегда можно выполнить, переименовав, если нужно, связанную переменную. Таким образом, всегда можно представлять Γ как конечную функцию, переводящую переменные в их типы. Согласно этому интуитивному представлению, множество переменных, присутствующих в Γ , можно обозначить выражением $\text{dom}(\Gamma)$.

Правило для типизации абстракций в общем случае имеет вид:

$$\frac{\Gamma, x:T_1 \vdash t_2:T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

где предпосылка добавляет ещё одно предположение к тем предположениям, что уже перечислены в заключении правила.

Правило типизации для переменных также немедленно следует из этого обсуждения: тип переменной попросту таков, как мы сейчас предполагаем.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

Предпосылка $x:T \in \Gamma$ означает: «Согласно Γ , предполагается, что x имеет тип T ».

Наконец, нужно сформулировать правило типизации для применений.

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$$

Если t_1 дает при вычислении функцию, переводящую аргументы типа T_{11} в результаты типа T_{12} (исходя из предположения, что значения, обозначаемые его свободными переменными, имеют типы, заданные в Γ), и если t_2 дает при вычислении результат типа T_{11} , то результат применения t_1 к аргументу t_2 будет иметь тип T_{12} .

Правила типизации для булевских констант и условных выражений остаются без изменений (рис. 8.1). Заметим, однако, что метапеременная T в правиле для условных выражений:

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$$

теперь может замещаться любыми функциональными типами. Это позволяет типизировать условные выражения, ветви которых содержат функции:¹

```
if true then ( $\lambda x:\text{Bool}. x$ ) else ( $\lambda x:\text{Bool}. \text{not } x$ );
 $\Rightarrow (\lambda x:\text{Bool}. x) : \text{Bool} \rightarrow \text{Bool}$ 
```

Все эти правила типизации сведены вместе на рис. 9.1 (и дополнены описанием синтаксиса и правил вычисления). Серым цветом на рисунке выделен материал, который отличает новую систему от бестипового лямбда-исчисления — как новые правила, так и фрагменты, добавленные к старым. Как и в случае с булевскими значениями и числами, мы разбили определение исчисления на две части: *чистое* (pure) простое типизированное лямбда-исчисление, где вообще нет никаких базовых типов (оно изображено на этом рисунке), и отдельный набор правил для булевских значений, который мы уже видели на рис. 8.1 (разумеется, к каждому правилу типизации на этом рисунке нужно добавить контекст Γ).

¹Начиная с этого момента, в примерах простого взаимодействия с интерпретаторами обычно будут показаны не только результаты, но и их типы (иногда мы будем их пропускать, если они очевидны).

\rightarrow (типовизированное)

Основано на λ (5.3)

Синтаксис	термы:	Вычисление	$t \rightarrow t'$
$t ::=$			
x	переменная	$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2}$	(E-APP1)
$\lambda x : T . t$	абстракция	$\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2}$	(E-APP2)
$t \ t$	применение		
$v ::=$	значения:		
$\lambda x : T . t$	значение-абстракция		
$T ::=$	типы:		
$T \rightarrow T$	тип функций		
$\Gamma ::=$	контексты:		
\emptyset	пустой контекст		
$\Gamma, x:T$	связывание термовой переменной		
		$(\lambda x : T_{11} . t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12}$	(E-APPABS)
			$[\Gamma \vdash t : T]$
		$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$	(T-VAR)
		$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1 . t_2 : T_1 \rightarrow T_2}$	(T-ABS)
		$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}}$	(T-APP)

Рис. 9.1. Чистое простое типизированное лямбда-исчисление (λ_\rightarrow)

Часто для обозначения простого типизированного лямбда-исчисления мы будем использовать символ λ_\rightarrow (мы будем использовать один символ для систем с разными наборами базовых типов).

Упражнение 9.2.1 [★]: На самом деле, чистое простое типизированное лямбда-исчисление без базовых типов является вырожденным (degenerate), в том смысле, что в нем нет ни одного корректно типизированного терма. Почему?

Из экземпляров правил типизации для λ_\rightarrow можно конструировать деревья вывода (derivation trees), как уже было сделано для арифметических выражений. Например, вот вывод, показывающий, что терм $(\lambda x:\text{Bool}.x) \ \text{true}$ имеет

типа Bool в пустом контексте.

$$\frac{\frac{x:\text{Bool} \in x:\text{Bool}}{x:\text{Bool} \vdash x:\text{Bool}} \text{-VAR} \quad \frac{}{\vdash \lambda x:\text{Bool}. x : \text{Bool} \rightarrow \text{Bool}} \text{T-Abs} \quad \frac{}{\vdash \text{true} : \text{Bool}} \text{T-TRUE}}{\vdash (\lambda x:\text{Bool}. x) \text{ true} : \text{Bool}} \text{T-APP}$$

Упражнение 9.2.2 [$\star \rightarrow$]: Покажите (путем построения деревьев вывода), что следующие термы имеют заявленные типы:

1. $f:\text{Bool} \rightarrow \text{Bool} \vdash f \ (\text{if } \text{false} \text{ then } \text{true} \text{ else } \text{false}) : \text{Bool}$
2. $f:\text{Bool} \rightarrow \text{Bool} \vdash \lambda x:\text{Bool}. f \ (\text{if } x \text{ then } \text{false} \text{ else } x) : \text{Bool} \rightarrow \text{Bool}$

Упражнение 9.2.3 [\star]: Найдите контекст Γ , в котором терм $f \ x$ у имеет тип Bool . Можете ли вы кратко описать множество всех таких контекстов?

9.3. Свойства типизации

Как и в главе 8, прежде, чем доказывать типовую безопасность, нужно установить несколько простых лемм. По большей части они похожи на те, что мы уже видели ранее — нужно просто добавить к отношению типизации контексты и в каждое доказательство ввести пункты, относящиеся к λ -абстракциям, применениям и переменным. Единственное действительно новое требование — это лемма о подстановке (substitution lemma) (9.3.8) для отношения типизации.

Прежде всего, лемма об инверсии (inversion lemma) содержит набор наблюдений о том, как строятся деревья вывода типов: пункт, относящийся к каждой синтаксической форме терма, говорит, что «если терм данной формы корректно типизирован, то его подтермы должны иметь типы такой-то формы...».

Лемма 9.3.1 [Инверсия отношения типизации]:

1. Если $\Gamma \vdash x : R$, то $x:R \in \Gamma$.
2. Если $\Gamma \vdash \lambda x:T_1. t_2 : R$, то $R = T_1 \rightarrow R_2$ для некоторого R_2 , и $\Gamma, x:T_1 \vdash t_2 : R_2$.
3. Если $\Gamma \vdash t_1 \ t_2 : R$, то существует некоторый тип T_{11} , такой, что $\Gamma \vdash t_1 : T_{11} \rightarrow R$ и $\Gamma \vdash t_2 : T_{11}$.
4. Если $\Gamma \vdash \text{true} : R$, то $R = \text{Bool}$.
5. Если $\Gamma \vdash \text{false} : R$, то $R = \text{Bool}$.
6. Если $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, то $\Gamma \vdash t_1 : \text{Bool}$ и $\Gamma \vdash t_2, t_3 : R$.

Доказательство: Лемма непосредственно следует из определения отношения типизации.

Упражнение 9.3.2 [РЕКОМЕНДУЕТСЯ, ★ ★ ★]: Существуют ли такой контекст Γ и такой тип T , что $\Gamma \vdash x : T$? Если да, то приведите пример Γ и T , и постройте дерево вывода $\Gamma \vdash x : T$; если нет, то докажите это.

В §9.2 мы выбрали для нашего исчисления представление с явной типизацией, чтобы облегчить задачу проверки типов. При этом нам пришлось добавить аннотации типов — но только в определениях связанных переменных и нигде больше. В каком смысле этого «достаточно»? Один из ответов на этот вопрос дает теорема о *единственности типов* (uniqueness of types), которая утверждает, что правильно типизированные термы находятся в однозначном соответствии со своими деревьями вывода типов: по терму можно однозначно восстановить дерево его вывода (и, разумеется, наоборот). В сущности, соответствие настолько прямое, что нет почти никакой разницы между термом и его деревом вывода.

Теорема 9.3.3 [Единственность типов]: В любом заданном контексте типизации Γ терм t (в котором все свободные переменные лежат в области определения Γ) имеет не более одного типа. То есть, если терм типизируем, то тип у него только один. Более того, существует только одно дерево вывода на основе правил, порождающих отношение типизации.

Доказательство: УПРАЖНЕНИЕ. Доказательство, в сущности, настолько прямолинейно, что почти не заслуживает изложения; однако выписывание всех деталей служит хорошей тренировкой по «выстраиванию» доказательств, касающихся отношения типизации.

Во многих системах типов, с которыми мы встретимся позднее в книге, такое простое соответствие между термами и деревьями вывода отсутствует: один и тот же терм может обладать несколькими типами, и каждый из них можно обосновать несколькими деревьями вывода. В этих системах часто будет требоваться нетривиальная работа, чтобы показать, что из термов можно эффективно получить деревья вывода типов.

Приведенная ниже лемма о канонических формах показывает, какой вид могут иметь значения различных типов.

Лемма 9.3.4 [Канонические формы]:

1. Если v — значение типа Bool , то v — это либо true , либо false .
2. Если v — значение типа $T_1 \rightarrow T_2$, то $v = \lambda x : T_1 . t_2$.

Доказательство: Не представляет сложности. (Подобно доказательству леммы о канонических формах для арифметических выражений, 8.3.1.)

С помощью леммы о канонических формах мы можем доказать теорему о продвижении, аналогичную теореме 8.3.2. Утверждение теоремы нуждается в небольшой поправке: нас интересуют только *замкнутые* термы, без свободных переменных. Для открытых термов теорема неверна: терм $f \ \text{true}$ является нормальной формой, но не значением. Однако этот пример не означает, что

в языке имеется дефект, поскольку полные программы — термы, вычисление которых нас на самом деле интересует, — всегда замкнуты.

Теорема 9.3.5 [ПРОДВИЖЕНИЕ]: Пусть t — замкнутый, правильно типизированный терм (т. е., для некоторого типа T , $\vdash t : T$). Тогда либо t является значением, либо имеется терм t' , такой, что $t \rightarrow t'$.

Доказательство: Прямолинейная индукция по деревьям вывода типов. Варианты с булевскими константами и условными выражениями точно повторяют доказательство теоремы о продвижении для типизированных арифметических выражений (8.3.2). Вариант с переменной не может возникнуть (поскольку t замкнут). Вариант с абстракцией следует непосредственно, поскольку абстракции — это значения.

Единственный интересный случай — это применение, в котором $t = t_1 t_2$, причем $\vdash t_1 : T_{11} \rightarrow T_{12}$ и $\vdash t_2 : T_{11}$. По предположению индукции, t_1 либо является значением, либо может произвести шаг вычисления; то же самое верно и для t_2 . Если t_1 может сделать шаг, то к t применимо правило E-APP1. Если t_1 — значение, а t_2 может сделать шаг, то к t применимо правило E-APP2. Наконец, если и t_1 , и t_2 — значения, то, по лемме о канонических формах, t_1 имеет вид $\lambda x:T_{11}.t_{12}$, так что к t применимо правило E-APPABs.

Нашей следующей задачей будет доказательство того, что вычисление сохраняет типы. Для начала мы сформулируем пару «структурных лемм» об отношении типизации. Сами по себе они не особенно интересны, но в последующих доказательствах они помогут нам проводить некоторые полезные преобразования с деревьями вывода типов.

Первая структурная лемма утверждает, что элементы контекста можно переставлять как угодно, не изменения при этом множество утверждений о типах, выводимых из этого контекста. Напомним (с. 121), что все связывания в контексте должны иметь различные имена, и что при каждом добавлении к контексту нового связывания мы молчаливо предполагаем, что новое имя отличается от уже присутствующих в контексте (в случае необходимости мы можем применить Соглашение 5.3.4 и переименовать новую переменную).

Лемма 9.3.6 [ПЕРЕСТАНОВКА]: Если $\Gamma \vdash t : T$, и Δ представляет собой перестановку Γ , то $\Delta \vdash t : T$. Более того, глубина дерева вывода остается неизменной.

Доказательство: Несложная индукция по деревьям вывода типов.

Лемма 9.3.7 [ОСЛАБЛЕНИЕ]: Если $\Gamma \vdash t : T$, и $x \notin \text{dom}(\Gamma)$, то $\Gamma, x:S \vdash t : T$. Более того, глубина дерева вывода остается неизменной.

Доказательство: Несложная индукция по деревьям вывода типов.

При помощи этих вспомогательных лемм можно доказать ключевое свойство отношения типизации: правильная типизированность сохраняется при подстановке вместо переменных термов соответствующих типов. Подобные леммы играют настолько большую роль в доказательствах о безопасности языков программирования, что часто их просто называют «леммами о подстановке».

Лемма 9.3.8 [СОХРАНЕНИЕ ТИПОВ ПРИ ПОДСТАНОВКЕ]: Если $\Gamma, x:S \vdash t : T$ и $\Gamma \vdash s : S$, то $\Gamma \vdash [x \mapsto s]t : T$.

Доказательство: Индукция по глубине вывода утверждения $\Gamma, x:S \vdash t : T$. Для каждого данного вывода мы рассматриваем варианты последнего используемого правила типизации.² Наиболее интересные случаи — переменная и абстракция.

Вариант T-VAR: $t = z$

где $z:T \in (\Gamma, x:S)$

Требуется рассмотреть два подварианта, в зависимости от того, соединяет ли z с x , или это разные переменные. Если $z = x$, то $[x \mapsto s]z = s$.

Требуемый результат тогда $\Gamma \vdash s : S$, а это одна из предпосылок леммы. В противном случае, $[x \mapsto s]z = z$, и результат следует непосредственно.

Вариант T-ABS: $t = \lambda y:T_2. t_1$

$T = T_2 \rightarrow T_1$

$\Gamma, x:S, y:T_2 \vdash t_1 : T_1$

По соглашению 5.3.4 можно считать, что $x \neq y$ и $y \notin FV(s)$. Согласно лемме о перестановке для одного из поддеревьев, получаем $\Gamma, y:T_2, x:S \vdash t_1 : T_1$. По лемме об ослаблении для другого поддерева ($\Gamma \vdash s : S$) получаем $\Gamma, y:T_2 \vdash s : S$. Согласно предположению индукции, получается $\Gamma, y:T_2 \vdash [x \mapsto s]t_1 : T_1$. По правилу T-ABS: $\Gamma \vdash \lambda y:T_2. [x \mapsto s]t_1 : T_2 \rightarrow T_1$. Но это именно то утверждение, которое нам нужно, поскольку, по определению подстановки, $[x \mapsto s]t = \lambda y:T_2. [x \mapsto s]t_1$.

Вариант T-APP: $t = t_1 t_2$

$\Gamma, x:S \vdash t_1 : T_2 \rightarrow T_1$

$\Gamma, x:S \vdash t_2 : T_2$

$T = T_1$

Согласно предположению индукции, $\Gamma \vdash [x \mapsto s]t_1 : T_2 \rightarrow T_1$ и $\Gamma \vdash [x \mapsto s]t_2 : T_2$. По правилу T-APP, $\Gamma \vdash [x \mapsto s]t_1 [x \mapsto s]t_2 : T$, т. е., $\Gamma \vdash [x \mapsto s](t_1 t_2) : T$.

Вариант T-TRUE: $t = \text{true}$

$T = \text{Bool}$

В этом случае $[x \mapsto s]t = \text{true}$, и требуемый результат $\Gamma \vdash [x \mapsto s]t : T$ следует непосредственно.

Вариант T-FALSE: $t = \text{false}$

$T = \text{Bool}$

Аналогично.

Вариант T-IF: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

$\Gamma, x:S \vdash t_1 : \text{Bool}$

$\Gamma, x:S \vdash t_2 : T$

$\Gamma, x:S \vdash t_3 : T$

Три раза применяя предположение индукции, имеем

²Или, что равносильно, варианты возможной формы t , поскольку для каждого синтаксического конструктора существует ровно одно правило типизации.

$\Gamma \vdash [x \mapsto s]t_1 : \text{Bool}$
 $\Gamma \vdash [x \mapsto s]t_2 : T$
 $\Gamma \vdash [x \mapsto s]t_3 : T$,
и утверждение леммы следует по правилу Т-IF.

Используя лемму о подстановке, мы можем доказать вторую половину теоремы о типовой безопасности — то, что вычисление сохраняет корректность типизации.

Теорема 9.3.9 [СОХРАНЕНИЕ]: Если $\Gamma \vdash t : T$ и $t \rightarrow t'$, то $\Gamma \vdash t' : T$.

Доказательство: УПРАЖНЕНИЕ [РЕКОМЕНДУЕТСЯ, ★★]. Структура доказательства очень похожа на доказательство теоремы о сохранении для арифметических выражений (8.3.3), за исключением использования леммы о подстановке.

Упражнение 9.3.10 [РЕКОМЕНДУЕТСЯ, ★★]: В упражнении 8.3.6 мы исследовали свойство расширения субъекта (*subject expansion*) для простого исчисления, работающего с типизированными арифметическими выражениями. Выполняется ли это свойство для «функциональной части» простого типизированного лямбда-исчисления? А именно: допустим, что t не содержит условных выражений. Можно ли, исходя из $t \rightarrow t'$ и $\Gamma \vdash t' : T$, сделать вывод $\Gamma \vdash t : T$?

9.4. Соотношение Карри-Говарда

Конструктору типов « \rightarrow » соответствуют два вида правил:

1. *правило введения* (introduction rule) (T-ABS), показывающее, как элементы типа могут *создаваться*, и
2. *правило устранения* (elimination rule) (T-APP), показывающее, как элементы типа могут *использоваться*.

Если форма, которая вводит элемент типа (λ) , является непосредственным подтермом формы, устраниющей его (применение), получается редекс — место, в котором может произойти вычисление.

Рассуждения о правилах введения и устранения часто оказываются полезными при исследовании систем типов. Позже, когда мы доберемся до более сложных систем, мы увидим, что у каждого конструктора типов есть аналогичные, связанные с ним, правила введения и устранения.

Упражнение 9.4.1 [★]: Какие из правил для типа *Bool* на рис. 8.1 являются правилами введения, а какие — правилами устранения? А для правил для типа *Nat* на рис. 8.2?

Терминология «правила введения/устранения» происходит из соответствия между теорией типов и логикой, известного как *соотношение Карри-Говарда* (Curry-Howard correspondence) или *изоморфизм Карри-Говарда* (Curry-Howard isomorphism) (Curry and Feys, 1958; Howard, 1980). Вкратце,

идея состоит в том, что в конструктивных логиках доказательство утверждения P состоит в демонстрации конкретного *свидетельства* (evidence) в пользу P .³ Карри и Говард заметили, что это свидетельство во многом похоже на вычисление. Например, доказательство утверждения $P \supset Q$ можно рассматривать как механическую процедуру, которая, получая доказательство P , строит доказательство Q — или, с другой точки зрения, доказательство Q , *абстрагированное* от доказательства P . Аналогично, доказательство $P \wedge Q$ состоит из доказательства P в сочетании с доказательством Q . Такое наблюдение приводит к следующему соответствуию:

Логика	Языки ПРОГРАММИРОВАНИЯ
утверждения	типы
утверждение $P \supset Q$	тип $P \rightarrow Q$
утверждение $P \wedge Q$	тип $P \times Q$ (см. §11.6)
доказательство утверждения P	терм t типа P
утверждение P доказуемо	существуют термы типа P

С этой точки зрения, терм в простом типизированном лямбда-исчислении является доказательством логического утверждения, соответствующего его типу. Вычисление (редукция на лямбда-термах) соответствует логической операции упрощения доказательств методом *устранения сечений* (cut elimination). Соотношение Карри-Говарда называют также аналогией «*утверждения как типы*» (*propositions as types*). Его подробное обсуждение можно найти во многих источниках, в том числе в книге Жирара, Лафонта и Тейлора (Girard, Lafont, and Taylor, 1989), у Галье (Gallier, 1993), Сёренсена и Ужичина (Sørensen and Urzyczyn, 1998), Пфеннинга (Pfenning, 2001), Губо-Ларрека и Макки (Goubault-Larrecq and Mackie, 1997), а также у Симмонса (Simmons, 2000).

Красота соотношения Карри-Говарда заключается в том, что оно не ограничено какой-то одной системой типов и одной логикой — напротив, его можно распространить на широкий спектр систем типов и логик. Например, Система F (гл. 23), в которой параметрический полиморфизм связан с квантификацией по типам, в точности соответствует конструктивной логике второго порядка, в которой разрешена квантификация по утверждениям. Система F_ω (гл. 30) соответствует логике высших порядков. Более того, это соответствие часто использовалось для переноса результатов из одной области в другую. Так, *линейная логика* (linear logic) Жирара (1987) приводит к идее *систем линейных типов* (linear type systems) (Wadler, 1990, Wadler, 1991, Turner, Wadler, and Mossin, 1995, Hodas, 1992, Mackie, 1994, Chirimar, Gunter, and Riecke, 1996, Kobayashi, Pierce, and Turner, 1996, и многие другие). Аналогично, *модальные логики* (modal logics) использовались при разработке систем *частичного вычисления* (partial evaluation) и *порождения кода во время выполнения* (run-

³Характерное различие между классическими и конструктивными логиками состоит в том, что в последних отсутствует *правило исключенного третьего* (excluded middle), которое утверждает, что для всякого утверждения Q истинно либо само Q , либо $\neg Q$. Чтобы доказать $Q \vee \neg Q$ в конструктивной логике, требуется предоставить свидетельство либо в пользу Q , либо в пользу $\neg Q$.

time code generation) (см. Davies and Pfenning, 1996, Wickline, Lee, Pfenning, and Davies, 1998, и другие источники, цитируемые в указанных трудах).

9.5. Стирание типов и типизируемость

На рис. 9.1 мы определили отношение вычисления непосредственно на просто типизированных термах. Несмотря на то, что аннотации типов не играют при вычислении никакой роли — во время выполнения мы не проводим никаких проверок того, что функции применяются к аргументам подходящих типов, — мы сохраняем эти аннотации внутри вычисляемых термов.

Большинство компиляторов промышленных языков программирования избегают сохранять аннотации во время выполнения: они используются при проверке типов (и, в более сложных компиляторах, при порождении кода), однако в скомпилированной программе их нет. В сущности, перед выполнением программы преобразуются обратно в бестиповую форму. Такой стиль семантики можно формализовать при помощи функции *стирания* (erasure), которая переводит типизированные термы в соответствующие бестиповые термы.

Определение 9.5.1 Функция стирания просто типизированного терма t определяется так:

$$\begin{array}{rcl} \text{erase}(x) & = & x \\ \text{erase}(\lambda x : T_1 . t_2) & = & \lambda x . \text{erase}(t_2) \\ \text{erase}(t_1 \ t_2) & = & \text{erase}(t_1) \text{ erase}(t_2) \end{array}$$

Разумеется, мы ожидаем, что два способа представления семантики просто типизированного лямбда-исчисления совпадут: результат вычисления типизированного терма напрямую не должен отличаться от результата вычисления бестипового терма, получившегося в результате предварительного стирания типов. Это требование формализуется в следующей теореме, которая формализует идею того, что «вычисление коммутирует со стиранием». Здесь имеется в виду, что эти операции можно проводить в любом порядке — вычислив, и потом стерев типы, мы получаем тот же самый результат, как если бы сначала стерли типы, а потом вычислили терм:

Теорема 9.5.2

1. Если $t \rightarrow t'$ согласно типизированному отношению вычисления, то $\text{erase}(t) \rightarrow \text{erase}(t')$.
2. Если $\text{erase}(t) \rightarrow m'$ согласно бестиповому отношению вычисления, то существует типизированный терм t' , такой, что $t \rightarrow t'$ и $\text{erase}(t') = m'$.

Доказательство: Несложная индукция по деревьям вывода вычисления.

Поскольку рассматриваемая нами «компиляция» настолько проста, теорема 9.5.2 очевидна до тривиальности. Однако для более интересных языков и более интересных компиляторов она становится важным свойством: она утверждает, что «высокоуровневая» семантика, напрямую выраженная на языке, используемом программистом, совпадает с альтернативной низкоуровневой стратегией вычисления, реально используемой в реализации языка.

Еще один интересный вопрос, связанный с функцией стирания, таков: если у нас есть бестиповый терм m , можем ли мы найти простой типизированный терм t , дающий при стирании типов m ?

Определение 9.5.3 Терм t бестипового лямбда-исчисления называется типизируемым (*typable*) в λ_{\rightarrow} , если имеется простой типизированный терм t типа T и контекст Γ такой, что $\text{erase}(t) = m$ и $\Gamma \vdash t : T$.

К этому вопросу мы вернемся в главе 22, когда рассмотрим близкородственную проблематику *реконструкции типов* (type reconstruction) для λ_{\rightarrow} .

9.6. Стиль Карри и стиль Чёрча

Как мы видели, семантику простого типизированного лямбда-исчисления можно определить двумя способами: как отношение вычисления, определенное прямо на синтаксисе простого типизированного исчисления, либо как компиляцию в бестиповое исчисление в сочетании с отношением вычисления на бестиповых термах. Важное сходство между этими двумя стилями состоит в том, что можно говорить о поведении терма t вне зависимости от того, корректно ли он типизирован. Такая форма определения языка называется *стилем Карри* (Curry-style). Сначала мы задаем грамматику термов, затем определяем их поведение, и, наконец, добавляем систему типов, отвергающую некоторые термы, поведение которых нам не нравится. Семантика возникает раньше, чем типизация.

Принципиально другой способ организации определения языка состоит в том, чтобы сначала определить термы, затем выбрать из них правильно типизированные термы, и, наконец, определить семантику только для них. В таких системах, которые называются системами *в стиле Чёрча* (Church-style), типизация идет прежде семантики: мы даже не задаём вопрос: «каково поведение неверно типизированного терма?». В сущности, строго говоря, в системах, построенных по Чёрчу, мы вычисляем не термы, а деревья вывода типов для них. (Пример можно найти в §15.6.)

Исторически сложилось, что неявно типизированные представления лямбда-исчислений часто описывают в стиле Карри, а представления Чёрча в основном встречаются для явно типизированных систем. Отсюда происходит некоторая путаница в терминологии: иногда «стилем Чёрча» называют явно типизированный *синтаксис*, а «стилем Карри» — неявно типизированный.

9.7. Дополнительные замечания

Простое типизированное лямбда-исчисление изучается в книге Хиндли и Селдина (Hindley and Seldin, 1986) и, еще более подробно, в монографии Хиндли (Hindley, 1997).

Правильно типизированные
программы не могут
«сломаться».

Робин Милнер (Milner 1978)

Глава 10

Реализация простых типов на ML

Конкретная реализация $\lambda\rightarrow$ в виде программы на ML следует той же схеме, что и реализация бестипового лямбда-исчисления в главе 7. Основное добавление — функция `typeof` для вычисления типа данного терма в данном контексте. Впрочем, прежде чем мы до нее доберемся, нам требуется написать несколько низкоуровневых процедур для работы с контекстами.

10.1. Контексты

Напомним (глава 7, с. 103), что контекст (`context`) представляет собой просто список имен переменных и связываний:

```
type context = (string * binding) list
```

В главе 7 контексты использовались исключительно для преобразования между именованной и безымянной формами термов при их считывании и распечатке. Для этого было достаточно знать только имена переменных; `binding` был определен в виде тривиального типа данных с одним конструктором, не несущим никакой информации:

```
type binding = NameBind
```

При реализации процедуры проверки типов нам понадобится использовать контекст для хранения сведений о типах переменных. Ради этого мы добавляем к типу `binding` новый конструктор `VarBind`:

```
type binding =
  NameBind
  | VarBind of ty
```

Реализация, описанная в этой главе, соответствует простому типизированному лямбда-исчислению (рис. 9.1) с булевскими значениями (8.1). Код из этой главы можно найти в репозитории под именем `simplebool`.

Каждый экземпляр с конструктором `VarBind` содержит сведения о типе соответствующей переменной. Помимо конструктора `VarBind`, мы сохраняем старый конструктор `NameBind` ради удобства функций чтения и распечатки, которым сведения о типе не нужны. (При другой стратегии реализации можно было бы определить два различных типа `context` — один для чтения и распечатки, а другой для проверки типов.)

Функция `typeof` вызывает функцию `addbinding`, чтобы добавить в контекст `ctx` новое связывание переменной (`x, bind`); поскольку контексты у нас представляются в виде списков, то `addbinding`, в сущности, работает просто как `cons`:

```
let addbinding ctx x bind = (x,bind)::ctx
```

Напротив, функция `getTypeFromContext` используется для поиска сведений о типе, связанном с некоторой переменной `i` в контексте `ctx` (информация `fi` о позиции в файле служит для распечатки сообщения об ошибке, когда `i` оказывается вне контекста):

```
let getTypeFromContext fi ctx i =
  match getbinding fi ctx i with
    VarBind(tyT) → tyT
  | _ → error fi
    ("getTypeFromContext: Wrong kind of binding for variable"
     ^ (index2name fi ctx i))
```

Оператор `match` производит проверку на внутреннюю непротиворечивость: в нормальных условиях `getTypeFromContext` должна всегда вызываться из контекста, где `i`-е связывание создано конструктором `VarBind`. Однако в последующих главах мы введем другие типы связываний (в частности, связывания для *типовых переменных*, type variables), и может случиться так, что `getTypeFromContext` будет вызвана с переменной неправильного вида. В этом случае она печатает сообщение об ошибке при помощи низкоуровневой функции `error`, передавая ей `info`, чтобы сообщить, в какой позиции в файле произошла ошибка.

```
val error : info → string → 'a
```

Тип результата функции `error` — типовая переменная `'a`, которая может принимать значение любого типа ML (что имеет смысл, так как `error` все равно никогда не возвращается: она печатает ошибку и останавливает программу). В данном случае следует предположить, что результат `error` имеет тип `ty`, так как именно его возвращает другая ветвь `match`.

Заметим, что информацию о типах мы ищем по *индексу*, поскольку внутри программы термы представляются в безымянной форме, и переменные в них представляются числовыми индексами. Функция `getbinding` ищет *i*-ое связывание в данном контексте:

```
val getbinding : info → context → int → binding
```

Ее определение можно найти в реализации `simplebool` на веб-сайте книги.

10.2. Термы и типы

Синтаксис типов прямо переводится из абстрактного синтаксиса, представленного на рис. 8.1 и 9.1, в определение типа языка ML.

```
type ty =
  TyBool
  | TyArr of ty * ty
```

Представление термов такое же, как было у нас при реализации бестипового лямбда-исчисления (с. 101), но с добавлением аннотации типа к варианту TmAbs.

```
type term =
  TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
```

10.3. Проверка типов

Функцию проверки типов `typeof` можно рассматривать как простой перевод правил типизации для λ_{\rightarrow} (рис. 8.1 и 9.1), или, точнее, как перевод леммы об инверсии (9.3.1). Второй подход точнее, поскольку именно лемма об инверсии определяет для каждой синтаксической формы, какие условия должны выполняться, чтобы терм считался правильно типизированным. Правила типизации утверждают, что термы некоторого вида правильно типизированы при некоторых условиях, но глядя на каждое конкретное правило типизации, никогда нельзя заключить, что некоторый терм *не* типизирован правильно, поскольку всегда остается возможность, что для типизации терма достаточно применить какое-то другое правило. (В данный момент эта разница может показаться несущественной, поскольку лемма об инверсии непосредственно следует из правил типизации. Однако это различие становится важным в последующих системах, в которых доказательство леммы об инверсии более трудоемко, чем в λ_{\rightarrow} .)

```
let rec typeof ctx t =
  match t with
    TmTrue(fi) →
      TyBool
    | TmFalse(fi) →
      TyBool
    | TmIf(fi,t1,t2,t3) →
      if (=) (typeof ctx t1) TyBool then
        let tyT2 = typeof ctx t2 in
        if (=) tyT2 (typeof ctx t3) then tyT2
        else error fi "arms of conditional have different types"
      else error fi "guard of conditional not a boolean"
```

```

| TmVar(fi,i,_) → getTypeFromContext fi ctx i
| TmAbs(fi,x,tyT1,t2) →
    let ctx' = addbinding ctx x (VarBind(tyT1)) in
    let tyT2 = typeof ctx' t2 in
    TyArr(tyT1, tyT2)
| TmApp(fi,t1,t2) →
    let tyT1 = typeof ctx t1 in
    let tyT2 = typeof ctx t2 in
    (match tyT1 with
     TyArr(tyT11,tyT12) →
     if (=) tyT2 tyT11 then tyT12
     else error fi "parameter_type_mismatch"
     | _ → error fi "arrow_type_expected")

```

Здесь полезно сделать несколько замечаний о языке OCaml. Во-первых, мы записываем OCaml-овскую операцию проверки на равенство = в скобках, потому что используем её в префиксной позиции, а не в нормальной инфиксной. Это делается для того, чтобы легче было сравнивать наш код с последующими версиями `typeof`, в которых операция сравнения типов будет более изощренной, чем простое сравнение. Во-вторых, операция сравнения проверяет *структурное* равенство составных значений, а не равенство *указателей*. А именно, выражение

```

let t  = TmApp(t1,t2) in
let t' = TmApp(t1,t2) in
(=) t t'

```

всегда возвращает `true`, несмотря на то, что два экземпляра `TmApp`, именуемые переменными `t` и `t'`, порождаются в разное время и имеют разные адреса в памяти.

Глава 11

Простые расширения

Простое типизированное лямбда-исчисление имеет достаточно сложную структуру, чтобы было интересно изучать его теорию, но оно не слишком похоже на нормальный язык программирования. В этой главе мы начинаем двигаться в сторону более привычных языков и добавляем к исчислению несколько привычных языковых конструкций, которые можно типизировать без особого труда. Важной темой на протяжении всей главы являются *производные формы* (derived forms).

11.1. Базовые типы

Всякий язык программирования предоставляет набор *базовых типов* (base types) — множеств простых, неструктурированных значений, например, чисел, булевских значений или символов, а также соответствующие элементарные операции для работы с этими значениями. Мы уже детально рассмотрели натуральные числа и булевские значения; проектировщик языка может точно таким же образом добавить в него столько дополнительных типов, сколько захочет.

Помимо типов `Bool` и `Nat`, мы иногда будем для оживления примеров использовать базовые типы `String` (строки, например "hello") и `Float` (числа с плавающей точкой, например 3.14159).

В теоретических целях часто бывает удобно абстрагироваться от свойств конкретных базовых типов и их операций, и вместо этого полагать, что язык снабжен некоторым множеством \mathcal{A} *неинтерпретируемых* (uninterpreted), или *неизвестных* (unknown), базовых типов, для которых не определено вообще никаких элементарных операций. Этого можно достигнуть, попросту включив \mathcal{A} в множество типов (с метапеременной A , которая принимает значения из \mathcal{A}), как показано на рис. 11.1. Для базовых типов мы используем букву A , а не B , чтобы избежать путаницы с символом \mathbb{B} , который означает присутствие булевых значений в данной системе. Можно считать, что буква A

В этой главе рассматриваются различные расширения чистого типизированного лямбда-исчисления (рис. 9.1). Соответствующая реализация на OCaml, `fullsimple`, содержит все эти расширения.

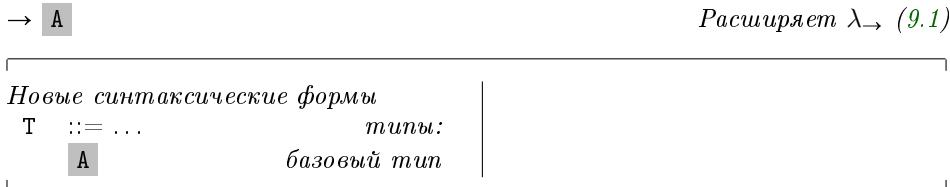


Рис. 11.1. Неинтерпретируемые базовые типы

отсылает к *атомарным* (atomic) типам — это название также часто используют для базовых типов, поскольку с точки зрения системы типов, у них нет никакой внутренней структуры. В качестве имен базовых типов мы будем употреблять символы A, B, C и т. д. Заметим, что, как и ранее с именами переменных и именами типовых переменных, A используется и как имя базового типа, и как метапеременная, принимающая базовые типы в качестве значений. Что конкретно имеется в виду в каждом случае, ясно из контекста.

Неужели от неинтерпретируемого типа нет никакой пользы? Вовсе нет. Хотя мы никак не можем прямо назвать его элементы, мы можем вводить переменные, принимающие значения этого типа. Например, функция¹

```
 $\lambda x : A . \quad x ;$ 
▷ <fun> : A → A
```

есть функция тождества для элементов A, каковы бы они ни были. Аналогично,

```
 $\lambda x : B . \quad x ;$ 
▷ <fun> : B → B
```

есть функция тождества для B, а

```
 $\lambda f : A \rightarrow A , \quad \lambda x : A . \quad f(f(x)) ;$ 
▷ <fun> : (A → A) → A → A
```

есть функция, дважды применяющая некоторую данную функцию f к аргументу x.

11.2. Единичный тип

Еще один полезный базовый тип, встречающийся в основном в языках семейства ML — единичный тип *Unit*, описанный на рис. 11.2. В отличие от неинтерпретируемых базовых типов из предыдущего раздела, этот тип интерпретируется наипростейшим образом: мы явно вводим единственный его элемент: термовую константу *unit* (пишется с маленькой буквы u), и правило типизации, превращающее *unit* в элемент *Unit*. Кроме того, мы добавляем *unit* к списку значений, могущих служить результатом вычисления: в сущности, *unit* — *единственный* возможный результат вычисления выражения типа *Unit*.

¹ Начиная с этого места, при записи результатов вычислений мы будем для экономии места заменять тела λ-абстракций на запись <fun>.

$\rightarrow \text{Unit}$

Расширяем $\lambda \rightarrow$ (9.1)

Новые синтаксические формы	
$t ::= \dots$	термы:
unit	константа unit
$v ::= \dots$	значения:
unit	константа unit
$T ::= \dots$	типы:
Unit	единичный тип

Новые правила типизации $\Gamma \vdash t : T$

$\Gamma \vdash \text{unit} : \text{Unit}$ (T-UNIT)

Новые производные формы

$$t_1; t_2 \stackrel{\text{def}}{=} (\lambda x:\text{Unit}. t_2) \ t_1 \quad \text{где } x \notin FV(t_2)$$

Рис. 11.2. Единичный тип

Даже в чисто функциональном языке тип **Unit** не лишен некоторого интереса,² однако в основном он применяется в языках с побочными эффектами, вроде присваиваний ссылочным ячейкам (к этой теме мы еще вернемся в главе 13). В таких языках нас часто интересует в выражении не результат вычисления, а его побочный эффект; для подобных выражений **Unit** служит подходящим типом результата.

Это применение **Unit** аналогично роли типа **void** в языках, подобных С и Java. Имя **void** наводит на мысль о пустом типе **Bot** (ср. §15.4), но используется он скорее как наш **Unit**.

11.3. Производные формы: последовательное исполнение и связывания-пустышки

В языках с побочными эффектами часто бывает полезно вычислить два выражения одно за другим. Конструкция *последовательного исполнения* (sequencing notation) $t_1; t_2$ означает: «вычислить t_1 , игнорировать его (триальный) результат, затем вычислить t_2 ».

Формализовать последовательное исполнение можно двумя способами. Во-первых, можно следовать той же стратегии, которую мы использовали для других синтаксических форм: добавить $t_1; t_2$ в качестве новой альтернативы к синтаксису термов, а затем ввести два правила вычисления

$$\frac{t_1 \rightarrow t'_1}{t_1; t_2 \rightarrow t'_1; t_2} \quad (\text{E-SEQ})$$

²Мы надеемся, что читателю доставит удовольствие следующая небольшая загадка:

Упражнение 11.2.1 [★ ★ ★]: Есть ли способ построить последовательность термов t_1, t_2, \dots на языке простого типизированного лямбда-исчисления с **Unit** в качестве единственного базового типа так, чтобы для каждого n терм t_n имел размер не более $O(n)$, но для достижения нормальной формы требовал не менее $O(2^n)$ шагов вычисления?

$$v_1; t_2 \rightarrow t_2 \quad (\text{E-SEQNEXT})$$

и правило типизации

$$\frac{\Gamma \vdash t_1 : \text{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 ; t_2 : T_2} \quad (\text{T-SEQ})$$

и таким образом зафиксировать требуемое поведение оператора ; («точка с запятой»).

Другой способ формализации последовательного исполнения состоит в том, чтобы рассматривать $t_1; t_2$ просто как *сокращенную запись* (abbreviation) для терма $(\lambda x:\text{Unit}.t_2) t_1$, в которой для переменной x выбирается *новое* (fresh) имя — то есть, такое, которое не совпадает ни с одной из свободных переменных t_2 .

Интуитивно достаточно ясно, что с точки зрения программиста оба способа добавить в язык последовательное исполнение приводят к одному и тому же результату: высокогородовые правила типизации и вычисления можно *вывести* (derive), если рассматривать $t_1; t_2$ как сокращение для $(\lambda x:\text{Unit}.t_2) t_1$. Это соответствие можно доказать более формально, продемонстрировав, что и типизацию, и вычисление можно «менять местами» с раскрытием сокращения.

Теорема 11.3.1 [ПОСЛЕДОВАТЕЛЬНОЕ ИСПОЛНЕНИЕ КАК ПРОИЗВОДНАЯ ФОРМА]: Обозначим простое типизированное лямбда-исчисление, дополненное типом *Unit*, конструкцией последовательного исполнения и правилами E-SEQ, E-SEQNEXT и T-SEQ, символом λ^E (где буква E взята из выражения external language, «внешний язык»). Обозначим простое типизированное лямбда-исчисление, куда добавлен только тип *Unit*, символом λ^I (internal language, «внутренний язык»). Пусть имеется $e \in \lambda^E \rightarrow \lambda^I$, функция раскрытия сокращений (*term elaboration function*), которая переводит выражения с внешнего языка на внутренний, заменяя каждое вхождение конструкции $t_1; t_2$ на $(\lambda x:\text{Unit}.t_2) t_1$ с новой переменной x . Тогда для каждого терма t языка λ^E имеем

- $t \rightarrow_E t'$ тогда и только тогда, когда $e(t) \rightarrow_I e(t')$
- $\Gamma \vdash^E t : T$ тогда и только тогда, когда $\Gamma \vdash^I e(t) : T$

где отношения вычисления и типизации языков λ^E и λ^I помечены, соответственно, символами E и I .

Доказательство: каждое направление в обоих “тогда и только тогда” доказывается с помощью прямолинейной индукции по структуре t .

Теорема 11.3.1 оправдывает применение термина «производная форма» (derived form), поскольку она демонстрирует, что поведение конструкции последовательного исполнения с точки зрения вычисления и типизации может быть выведено из более базовых операций — абстракции и применения. Преимущество введения таких конструкций, как последовательное исполнение, в виде производных форм состоит в том, что мы таким образом расширяем

внешний синтаксис (т. е., язык, реально используемый программистами при написании программ), но при этом избегаем усложнения внутреннего языка, для которого нужно доказывать такие теоремы, как теорему о типовой безопасности. Такой метод борьбы со сложностью описания языковых конструкций можно найти уже в «Определении Algol 60» (Naur et al., 1963), и он широко применяется во многих более современных описаниях языков, таких, как «Определение Standard ML» (Milner, Tofte, and Harper, 1990; Milner, Tofte, Harper, and MacQueen, 1997).

Часто производные формы вслед за Ландином называют *синтаксическим сахаром* (syntactic sugar). Замена производной формы ее низкоуровневым определением называется *удалением сахара* (desugaring).

Еще одна производная форма, которой мы часто будем пользоваться впоследствии — соглашение о «связываниях-пустышках» в конструкциях, связывающих переменные. Часто бывает нужно (например, в термах, создаваемых при удалении сахара из конструкций последовательного исполнения) записать лямбда-абстракцию как «заглушку», в теле которой связываемая переменная нигде не встречается. В таких случаях бывает неудобно каждый раз придумывать новое имя; вместо этого мы будем заменять его *связыванием-пустышкой* (wildcard binder), которое выглядит как $_$ («подчерк»). Таким образом, мы будем пользоваться записью $\lambda_\mathbf{S}.\mathbf{t}$ в качестве сокращения записи $\lambda\mathbf{x}:\mathbf{S}.\mathbf{t}$, где \mathbf{x} — некоторая переменная, не встречающаяся в \mathbf{t} .

Упражнение 11.3.2 [★]: Запишите правила вычисления и типизации для абстракций со связыванием-пустышкой и докажите, что их можно вывести из правил раскрытия сокращений, приведенных выше.

11.4. Приписывание типа

Еще одна простая конструкция, которая впоследствии часто будет нам полезна, — это явное *приписывание* (ascription) определенного типа определенному терму (т. е., в программе явно записывается утверждение, что данный терм имеет данный тип). Мы пишем « \mathbf{t} as \mathbf{T} », имея в виду «терм \mathbf{t} , которому мы приписываем тип \mathbf{T} ». Правило типизации T-ASCRIBE для этой конструкции (см. рис. 11.3) просто-напросто проверяет, что \mathbf{T} действительно является типом терма \mathbf{t} . Правило вычисления E-ASCRIBE столь же очевидно: оно отбрасывает as-конструкцию, и \mathbf{t} после этого вычисляется как обычно.

Приписывание типов может быть полезно в нескольких ситуациях. Часто оно служит для *документирования*. Иногда читателю бывает трудно уследить за типами всех подвыражений в сложном составном выражении. Разумное использование явных приписываний типа может заметно облегчить задачу. Более того, в особенно сложных выражениях даже *автору* может быть неясно, каковы типы всех подвыражений. Приписывание типов нескольким из них может помочь программисту прояснить собственные мысли. И в самом деле, иногда явное приписывание типов помогает найти источник трудноуловимых ошибок типизации.

Еще одно использование явного приписывания типов — управление *распечаткой* (printing) сложных типов. Программы проверки типов, которые ис-

\rightarrow asРасширяет $\lambda \rightarrow$ (9.1)

Новые синтаксические формы	Новые правила типизации
$t ::= \dots$	$\Gamma \vdash t : T$
$t \ as \ T$ приписывание типа	$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \ as \ T : T} \quad (\text{T-ASCRIBE})$
Новые правила вычисления	$t \rightarrow t'$
$v_1 \ as \ T \rightarrow v_1$	(E-ASCRIBE)
$\frac{t_1 \rightarrow t'_1}{t_1 \ as \ T \rightarrow t'_1 \ as \ T} \quad (\text{E-ASCRIBE1})$	

Рис. 11.3. Приписывание типа

пользованы для проверки примеров в этой книге, а также вспомогательные реализации на OCaml, имена которых начинаются с `full` («полный»), обеспечивают простой механизм для введения сокращений вместо длинных выражений типов. (В остальных реализациях механизм сокращений опущен, чтобы интерпретаторы легче было читать и модифицировать.) Например, объявление

```
UU = Unit → Unit;
```

превращает идентификатор `UU` в сокращение для типа `Unit → Unit`, которое можно использовать в программе. Где бы мы ни встретили идентификатор `UU`, его следует понимать как `Unit → Unit`. Например, можно написать

```
(λf:UU. f unit) (λx:Unit. x);
```

Во время проверки типов эти сокращения по необходимости раскрываются. С другой стороны, процедуры проверки типов пытаются по возможности эти определения свернуть. (А именно, каждый раз, проверяя подтерм, они смотрят, не совпадает ли его тип в точности с каким-нибудь из известных сокращений.) Обычно при этом получаются разумные результаты, но изредка нам требуется распечатывать тип как-то иначе: либо из-за того, что простая стратегия сопоставления с образцом не дает программе проверки типов использовать сокращение (скажем, в системах, где разрешается изменять порядок именованных полей типов записи, она не поймет, что `{a:Boolean, b:Nat}` можно свободно заменять на `{b:Nat, a:Boolean}`), либо по какой-нибудь другой причине. К примеру, в

```
λf:Unit → Unit. f;
▷ <fun> : (Unit → Unit) → UU
```

сокращение `UU` используется в результате функции, но не используется в ее аргументе. Если нам хочется, чтобы тип обозначался как `UU`, нужно либо изменить аннотацию типа на выражении-абстракции

```

λf:UU. f ;
> <fun> : UU → UU

```

либо явно приписать тип всей абстракции:

```

(λf:Unit → Unit. f) as UU → UU ;
> <fun> : UU → UU

```

Когда процедура проверки типов обрабатывает приписывание типа t *as* T , она разворачивает каждое сокращение в T , проверяя, что t имеет тип T , однако затем возвращает в качестве типа выражения-приписывания сам T , в частности так, как он был записан. Использование приписываний типа для управления печатью типов — особенность именно интерпретаторов, используемых в этой книге. В полноценном языке программирования механизмы для сокращения и распечатки типов либо не нужны (например, в Java все типы представляются короткими именами — см. в главе 19), либо намного глубже встроены в язык (как в OCaml — см. Rémy and Vouillon, 1998; Vouillon, 2000).

Наконец, приписывание типов можно использовать в качестве механизма *абстракции* (*abstraction*). Этот способ подробно обсуждается в §15.5. В системах, в которых один и тот же терм t может иметь несколько типов (скажем, в системах с подтипами), через приписывание можно «спрятать» некоторые из них, указав программе проверки, что t нужно рассматривать, как если бы он обладал меньшим набором типов. Кроме того, в §15.5 будет обсуждаться связь между явным приписыванием типов и их *преобразованием* (*casting*).

Упражнение 11.4.1 [РЕКОМЕНДУЕТСЯ, ★★]: (1) Покажите, как приписывание типов можно представить в виде производной формы. Докажите, что «официальные» правила типизации и вычисления должны образом соответствуют вашему определению. (2) Предположим, что вместо пары правил вычисления E-ASCRIBE и E-ASCRIBE1, мы ввели бы «энергичное» правило

$$t_1 \text{ as } T \rightarrow t_1 \quad (\text{E-ASCRIBEAGER})$$

немедленно отбрасывающее приписывание с каждого обнаруженного терма. Можно ли при этом по-прежнему считать приписывание типа производной формой?

11.5. Связывание let

При написании сложного выражения часто бывает полезно присвоить имя некоторым его подвыражениям (как для того, чтобы избежать повторений, так и для простоты чтения). В большинстве языков есть способы сделать это. Например, в ML запись `let x = t1 in t2` означает «вычислить выражение t_1 и связать имя x с получившимся значением при вычислении t_2 ».

Наша конструкция связывания через `let` (представленная на рис. 11.4), подобно ML, следует порядку вычисления с вызовом по значению. Это означает, что терм, связанный через `let`, должен быть полностью вычислен, прежде чем начнет вычисляться тело `let`-формы. Правило типизации T-LET указывает, что тип `let`-выражения можно получить, сначала определив тип связанного терма, расширив контекст связыванием с этим типом, и вычислив в этом

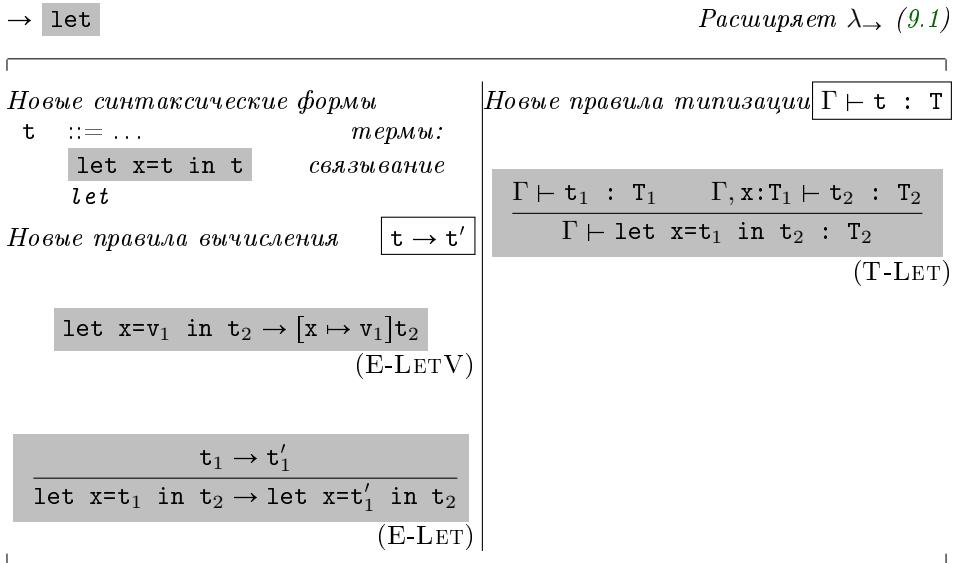


Рис. 11.4. Связывание let

расширенном контексте тип тела **let**, который и будет тогда типом всего **let**-выражения.

Упражнение 11.5.1 [РЕКОМЕНДУЕТСЯ, ★ ★ ★]: Программа проверки типов *letexercise* (ее можно скачать на сайте книги) является неполной реализацией *let*-выражений: в ней имеются простые функции синтаксического анализа и распечатки, но в функциях *eval1* и *typeof* нет вариантов для конструктора *TmLet* (на их месте находятся заглушки, которые срабатывают на любой терм, а при попытке исполнения завершают программу с сообщением об ошибке). Допишите программу.

Можно ли определить **let** в виде производной формы? Да, как показал Ландин; однако детали реализации несколько сложнее, чем для последовательного исполнения и приписывания типов. С наивной точки зрения ясно, что эффекта **let**-связывания можно достичь, сочетая абстракцию и применение:

$$\text{let } x=t_1 \text{ in } t_2 \stackrel{\text{def}}{=} (\lambda x:T_1 . t_2) \ t_1$$

Заметим, однако, что правая часть определения содержит аннотацию типа T_1 , которая в левой части отсутствует. То есть, если мы считаем, что производные формы подвергаются удалению сахара в процессе синтаксического анализа компилятором, то нам нужно понять, откуда синтаксический анализатор узнает, что в качестве аннотации типа в λ -выражении для обессахаренного терма на внутреннем языке он должен породить T_1 .

Ответ на этот вопрос, разумеется, таков: информация приходит из программы проверки типов! Требуемая аннотация типов получается путем вычисления типа терма t_1 . С более формальной точки зрения, это говорит о

тот, что конструктор `let` — производная форма несколько иного рода, чем виденные нами до сих пор: ее нужно рассматривать не как обессахаривающее преобразование термов, а как преобразование на *деревьях вывода типов* (или, если удобнее так считать, на термах, которые процедура проверки типов пометила результатами своего анализа). При этом дерево, включающее `let`

$$\frac{\vdots \quad \vdots}{\Gamma \vdash t_1 : T_1 \quad \Gamma, x:T_1 \vdash t_2 : T_2} \text{ T-LET}$$

$$\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2$$

переводится в дерево, включающее абстракцию и применение:

$$\frac{\vdots \quad \vdots}{\Gamma, x:T_1 \vdash t_2 : T_2} \text{ T-Abs}$$

$$\frac{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2 \quad \vdots \quad \vdots}{\Gamma \vdash (\lambda x:T_1. t_2) \ t_1 : T_2} \text{ T-APP}$$

$$\Gamma \vdash t_1 : T_1$$

Таким образом, `let` — «несколько менее производная» форма, чем уже виденные нами: ее поведение при вычислении можно выявить, удалив сахар, но ее поведение с точки зрения типизации должно быть встроено во внутренний язык.

В главе 22 мы встретим еще один довод против определения `let` в качестве производной формы: в языках с полиморфизмом по Хиндли-Милнеру (т. е., основанных на унификации) программа проверки типов обрабатывает `let`-формы особым образом и с их помощью *обобщает* (generalizes) полиморфные определения, получая при этом типы, которые нельзя смоделировать через обыкновенную λ -абстракцию и применение.

Упражнение 11.5.2 [★★]: *Можно попробовать удалить сахар в форме `let` путем ее немедленного «вычисления», т. е. рассматривать `let x=t1 in t2` как сокращение для $[x \mapsto t_1]t_2$. Насколько хороша эта идея?*

11.6. Пары

В большинстве языков программирования имеются различные способы построения составных структур данных. Простейшая из таких структур — *пары* (pairs), или, в более общем случае, *кортежи* (tuples) значений. В этом разделе мы рассматриваем пары, а более общие конструкции — кортежи и записи с метками полей — откладываем до §11.7 и §11.8.³

Формальное определение пар едва ли стоит подробного обсуждения — для читателя, добравшегося до этого места в книге, чтение правил на рис. 11.5 должно быть едва ли не проще, чем текст на естественном языке, содержащий ту же самую информацию. Давайте, однако, рассмотрим вкратце различные детали определения, чтобы подчеркнуть общую структуру.

³ В реализации `fullsimple` синтаксис пар, приводимый здесь, не поддерживается, поскольку кортежи все равно являются более общим решением.

\rightarrow \times

Расширяет λ_\rightarrow (9.1)

Новые синтаксические формы

$t ::= \dots$ *термы:*

$\{t, t\}$ *пара*

$t.1$ *первая проекция*

$t.2$ *вторая проекция*

$v ::= \dots$ *значения:*

$\{v, v\}$ *значение-пара*

$T ::= \dots$ *типы:*

$T_1 \times T_2$ *тип-произведение*

Новые правила вычисления $t \rightarrow t'$

$\{v_1, v_2\}.1 \rightarrow v_1$ (E-PAIRBETA1)

$\{v_1, v_2\}.2 \rightarrow v_2$ (E-PAIRBETA2)

$$\frac{t_1 \rightarrow t'_1}{t_1.1 \rightarrow t'_1.1}$$
 (E-PROJ1)

Новые правила типизации $\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2}$$
 (T-PAIR)

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}}$$
 (T-PROJ1)

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}}$$
 (T-PROJ2)

Рис. 11.5. Пары

Чтобы добавить пары к простому типизированному лямбда-исчислению, требуются два новых вида термов: порождение пары, которое записывается в виде $\{t_1, t_2\}$, и проекция, которая записывается в виде $t.1$, если это первая проекция, и в виде $t.2$, если это вторая проекция. Также необходим новый конструктор типов $T_1 \times T_2$, который называется *произведением* (product) (или, иногда, *декартовым произведением*, cartesian product) типов T_1 и T_2 . Пары записываются в фигурных скобках,⁴ чтобы подчеркнуть их связь с записями из §11.8.

⁴Способ записи с фигурными скобками не вполне удачен для пар и кортежей, поскольку напоминает стандартное математическое обозначение множеств. Чаще, как в популярных языках вроде ML, так и в исследовательской литературе, используются круглые скобки. Встречаются также и другие виды нотации, например, квадратные или угловые скобки.

В правилах вычисления нужно указать, как ведут себя пары и их проекции. Правила E-PAIRBETA1 и E-PAIRBETA2 указывают, что при сочетании полностью вычисленной пары с первой или второй проекцией результатом является соответствующая компонента пары. E-PROJ1 и E-PROJ2 позволяют производить вычисления внутри проекций, если терм, из которого производится проекция, еще не полностью вычислен. E-PAIR1 и E-PAIR2 вычисляют части пары: сначала левую, а затем — когда слева окажется значение, — правую часть.

Порядок, возникающий из использования метапеременных v и t в этих правилах, обеспечивает для пар стратегию вычисления слева направо. Например, составной терм

```
{pred 4, if true then false else false}.1
```

вычисляется (только) так:

```
{pred 4, if true then false else false}.1
→ {3, if true then false else false}.1
→ {3, false}.1
→ 3
```

Кроме того, нужно добавить новый вариант к определению значений, чтобы указать, что $\{v_1, v_2\}$ является значением. То, что компоненты пары-значения сами обязаны быть значениями, обеспечивает полное вычисление пары, передаваемой как аргумент функции, прежде, чем начнет выполняться тело функции. Например:

```
(λx:Nat×Nat. x.2) {pred 4, pred 5}
→ (λx:Nat×Nat. x.2) {3, pred 5}
→ (λx:Nat×Nat. x.2) {3,4}
→ {3,4}.2
→ 4
```

Правила типизации для пар и их проекций не представляют сложности. Правило введения T-PAIR указывает, что $\{t_1, t_2\}$ имеет тип $T_1 \times T_2$, при условии, что t_1 имеет тип T_1 , а t_2 — тип T_2 . Соответственно, правила устремления T-PROJ1 и T-PROJ2 говорят, что если t_1 имеет тип-произведение $T_{11} \times T_{12}$ (т. е., если этот терм даст в качестве значения пару), то типами проекций этой пары будут T_{11} и T_{12} .

11.7. Кортежи

Понятие двухместного произведения типов из предыдущего раздела нетрудно расширить до n -местных произведений, называемых *кортежами* (tuples). Например, $\{1, 2, \text{true}\}$ — трехместный кортеж, содержащий два числа и булевское значение. Его тип записывается как $\{\text{Nat}, \text{Nat}, \text{Bool}\}$.

Единственная трудность при таком обобщении состоит в том, что требуется придумать способ записи, единообразно описывающий структуры произвольного размера; такие соглашения всегда представляют некоторую сложность из-за неизбежного компромисса между точностью и читаемостью. Мы используем обозначение $\{t_i^{i \in 1..n}\}$ для кортежа из n членов, от t_1 до t_n , и $\{T_i^{i \in 1..n}\}$

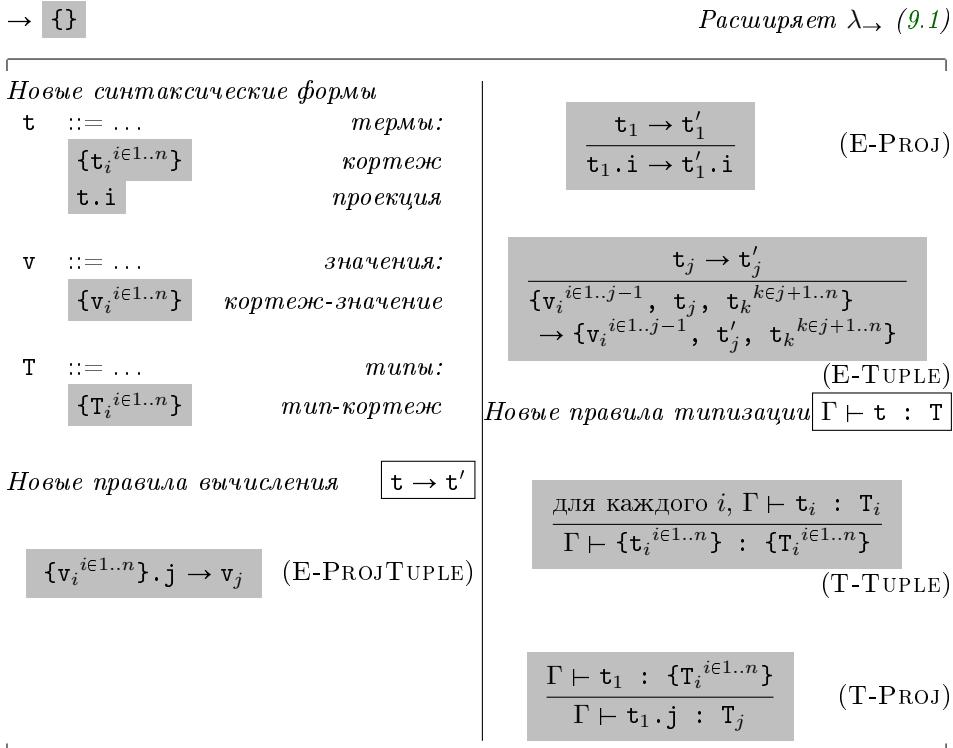


Рис. 11.6. Кортежи

для его типа. Заметим, что при этом n может равняться нулю; в таком случае, диапазон $1..n$ пуст, а $\{t_i^{i \in 1..n}\}$ равняется $\{\}$, пустому кортежу. Кроме того, обратите внимание на различие между простым значением вроде 5 и однозначным кортежем вроде {5}: единственная операция, которую можно произвести с последним — извлечение первой компоненты.

На рис. 11.6 дано формальное определение кортежей. Оно похоже на определение типов-произведений (рис. 11.5), но с обобщением каждого правила для пар на случай с n членами, причем каждая пара правил для первой и второй проекции превращается в единственное правило для произвольной проекции кортежа. Единственное правило, заслуживающее отдельного внимания — E-TUPLE, сочетающее и обобщающее правила E-PAIR1 и E-PAIR2 на рис. 11.5. На естественном языке его смысл выражается так: если имеется кортеж, в котором все поля слева от поля j уже преобразованы в значения, то можно проделать один шаг вычисления этого поля, из t_j в t'_j . Как и раньше, при помощи метапеременных мы обеспечиваем стратегию вычисления слева направо.

$\rightarrow \{\}$ Расширяем λ_\rightarrow (9.1)

Новые синтаксические формы

$t ::= \dots$	термы:
$\{l_i=t_i^{i \in 1..n}\}$	запись
$t.l$	проекция

$$\frac{t_1 \rightarrow t'_1}{t_1.l \rightarrow t'_1.l} \quad (\text{E-PROJ})$$

$v ::= \dots$	значения:
$\{l_i=v_i^{i \in 1..n}\}$	запись-значение

$$\frac{t_j \rightarrow t'_j}{\begin{aligned} &\{l_i=v_i^{i \in 1..j-1}, l_j=t_j, l_k=t_k^{k \in j+1..n}\} \\ &\rightarrow \{l_i=v_i^{i \in 1..j-1}, l_j=t'_j, l_k=t_k^{k \in j+1..n}\} \end{aligned}} \quad (\text{E-RCD})$$

$T ::= \dots$	типы:
$\{l_i:T_i^{i \in 1..n}\}$	тип записей

Новые правила типизации $\boxed{\Gamma \vdash t : T}$

Новые правила вычисления

 $t \rightarrow t'$

$$\{l_i=v_i^{i \in 1..n}\}.l_j \rightarrow v_j \quad (\text{E-PROJRCD})$$

$$\frac{\text{для каждого } i, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i=t_i^{i \in 1..n}\} : \{l_i:T_i^{i \in 1..n}\}} \quad (\text{T-RCD})$$

$$\frac{\Gamma \vdash t_1 : \{l_i:T_i^{i \in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \quad (\text{T-PROJ})$$

Рис. 11.7. Записи

11.8. Записи

Переход от n -местных кортежей к записям с помеченными полями также не представляет труда. Мы просто снабжаем каждое поле t_i меткой (label), выбираемой из некоторого заранее заданного множества \mathcal{L} . Например, `{x=5}` и `{partno=5524, cost=30.27}` — записи-значения; их типы — `{x:Nat}` и `{partno:Nat, cost:Float}`. Мы требуем, чтобы все метки полей в каждом конкретном терме-записи или типе записи были различны.

Правила для записей сведены в таблицу на рис. 11.7. Единственное правило, достойное отдельного упоминания, — E-PROJRCD, в котором мы опираемся на не вполне формальное соглашение. Это правило следует понимать так: если есть запись $\{l_i=v_i^{i \in 1..n}\}$ с меткой j -го поля l_j , то $\{l_i=v_i^{i \in 1..n}\}.l_j$ за один шаг переходит в j -ое значение поля, v_j . От этого соглашения (и от аналогичного соглашения для кортежей в правиле E-PROJTUPLE) можно было бы избавиться, переформулировав правило в более явном виде; однако при этом мы слишком сильно проиграли бы в удобстве чтения.

Упражнение 11.8.1 [★ →]: Запишите, для сравнения, E-PROJRCD в более явном виде.

Обратите внимание, что в верхнем углу блока правил, определяющих

кортежи и записи, стоит одна и та же «метка конструкции» `{}`. В самом деле, кортежи можно считать особым случаем записей, просто разрешив в качестве меток полей как натуральные числа, так и слова-идентификаторы. В случае, когда i -ое поле записи имеет метку i , мы эту метку опускаем. Например, `{Bool, Nat, Bool}` считается сокращенной записью для `{1:Bool, 2:Nat, 3:Bool}`. (Такое соглашение разрешает даже смешивать именованные поля с нумерованными, и писать `{a:Bool, Nat, c:Bool}` как сокращение `{a:Bool, 2:Nat, c:Bool}`, однако это вряд ли будет полезно на практике.) Во многих языках кортежи и записи считаются разными типами из более практических соображений: они по-разному реализуются компилятором.

Языки программирования различаются тем, как они относятся к порядку полей в записях. Во многих языках порядок полей как в значениях-записях, так и в определениях их типов никак не влияет на значение — т. е., термы `{partno=5524, cost=30.27}` и `{cost=30.27, partno=5524}` означают одно и то же и имеют один и тот же тип, который записывается либо в виде `{partno:Nat, cost:Float}`, либо в виде `{cost:Float, partno:Nat}`. Мы приняли другое решение: в этом тексте `{partno=5524, cost=30.27}` и `{cost=30.27, partno=5524}` представляют собой *разные* значения, с типами `{partno:Nat, cost:Float}` и `{cost:Float, partno:Nat}`, соответственно. В главе 15 мы примем другую, менее строгую, точку зрения на порядок полей, и введем отношение подтипов, в котором типы `{partno:Nat, cost:Float}` и `{cost:Float, partno:Nat}` будут *эквивалентны* (*equivalent*) — каждый из них будет подтипом другого, так что термы одного типа всегда можно будет использовать там, где ожидается значение другого. (При наличии подтипов выбор между неупорядоченными и упорядоченными записями серьезно влияет на производительность; это обсуждается далее, в §15.6. Если принять решение в пользу неупорядоченных записей, то выбор между вариантом, в котором записи считают неупорядоченными с самого начала, и вариантом, в котором поля упорядочены на низком уровне, но введены правила, позволяющие этот порядок игнорировать, становится делом вкуса. Мы выбрали второй вариант, чтобы получить возможность обсудить оба.)

Упражнение 11.8.2 [★★★]: В нашем описании записей операция проекции позволяет извлечь поля записи по одному. Многие высокоуровневые языки программирования обладают альтернативной синтаксической конструкцией сопоставления с образцом (*pattern matching*), при помощи которой все поля записи можно извлечь одновременно. Такая конструкция позволяет сильно сократить многие программы. Как правило, образцы могут вкладываться друг в друга, и таким образом можно легко извлекать данные из сложных вложенных структур данных.

Можно добавить простую разновидность сопоставления с образцом к бестиповому лямбда-исчислению, введя новую синтаксическую категорию образцов (*patterns*), а также новый вариант (для самой конструкции сопоставления) к синтаксису термов. (См. рис. 11.8.)

Правило вычисления для сопоставления с образцом является обобщенным случаем для правила связывания `let` с рис. 11.4. Оно опирается на вспомогательную «сопоставляющую» функцию, которая, принимая образец p и значение v , либо терпит неудачу (что означает, что значение v не соот-

$\rightarrow \{\} \text{ let } p \text{ (бестиповое)}$

Расширяет 11.7 и 11.4

Новые синтаксические формы

$p ::= x$ образец-переменная
 $\{l_i=p_i^{i \in 1..n}\}$ образец для записи

$t ::= \dots$ термы:
 $\text{let } p=t \text{ in } t$ связывание
 с образом

Правила сопоставления

$\text{match } (x, v) = [x \mapsto v]$ (M-VAR)

$$\frac{\text{для каждого } i, \quad \text{match } (p_i, v_i) = \sigma_i}{\text{match } (\{l_i=p_i^{i \in 1..n}\}, \{l_i=v_i^{i \in 1..n}\})} = \sigma_1 \circ \dots \circ \sigma_n$$

$$\frac{\text{Новые правила вычисления}}{t \rightarrow t'} \quad (\text{M-RCD})$$

$$\frac{\text{let } p=v_1 \text{ in } t_2 \rightarrow \text{match } (p, v_1) \quad t_2}{(\text{E-LETV})}$$

$$\frac{\text{let } p=t_1 \rightarrow t_1 \rightarrow \text{let } p=t'_1 \text{ in } t_2}{(\text{E-LET})}$$

Рис. 11.8. (Бестиповые) образцы для записей

вествует образцу), либо выдает в качестве результата подстановку, переводящую переменные, содержащиеся в p , в соответствующие части v . Например, $\text{match}(\{x, y\}, \{5, \text{true}\})$ порождает подстановку $[x \mapsto 5, y \mapsto \text{true}]$, сопоставление $\text{match}(x, \{5, \text{true}\})$ порождает $[x \mapsto \{5, \text{true}\}]$, а $\text{match}(\{x\}, \{5, \text{true}\})$ терпит неудачу. Правило E-LETV с помощью match вычисляет подстановку для переменных в p .

Сама функция match определяется отдельным набором правил вывода. Правило M-VAR утверждает, что сопоставление с переменной всегда завершается успешно, и возвращает подстановку, переводящую переменную в значение, с которым производится сопоставление. Правило M-RCD говорит, что чтобы сопоставить образец-запись $\{l_i=p_i^{i \in 1..n}\}$ со значением-записью $\{l_i=v_i^{i \in 1..n}\}$ (одинаковой длины, с одинаковыми метками полей), нужно по отдельности сопоставить каждый подобразец p_i с соответствующим подзначением v_i , получая при этом подстановку σ_i , и построить окончательный результат в виде композиции всех таких подстановок. (Мы требуем, чтобы ни одна переменная не встречалась в образце более одного раза, так что композиция подстановок будет просто их обединением.)

Покажите, как к этой системе добавить типы.

1. Введите правила типизации для новых конструкций (при этом можно вносить в синтаксис любые нужные изменения).
2. Постройте схематическое доказательство теорем о сохранении и приведении для всего полученного исчисления. (Строить полные доказательства необязательно; достаточно сформулировать требуемые леммы и расположить их в правильном порядке.)

11.9. Типы-суммы

Во многих программах требуется работать с *разнородными* (heterogeneous) наборами данных. К примеру, вершина в бинарном дереве может быть либо «листом», либо внутренней вершиной с двумя дочерними; аналогично, ячейка списка может быть либо пустой (`nil`), либо `cons`-ячейкой, состоящей из головы и хвоста;⁵ вершина абстрактного синтаксического дерева внутри компилятора может соответствовать переменной, абстракции, применению функции и т. д. Механизм теории типов, поддерживающий программирование такого рода, называется *типами-вариантами* (variant types).

Прежде чем ввести варианты в общем случае (в §11.10), рассмотрим более простой случай бинарных *типов-сумм* (sum types). Тип-сумма описывает множество значений, выбираемых из ровно двух данных типов. Допустим, например, что у нас есть типы

```
PhysicalAddr = {first last: String, addr: String};  
VirtualAddress = {name: String, email: String};
```

представляющие собой различные виды записей в адресной книге. Если мы хотим обрабатывать оба вида записей единообразно (например, если нам требуется построить список, содержащий записи обоих видов), можно ввести тип-сумму,⁶

```
Addr = PhysicalAddr + VirtualAddress;
```

каждый из элементов которого представляет собой либо `PhysicalAddr`, либо `VirtualAddress`.

Элементы этого типа создаются путем *постановки тегов* (tagging) на элементы типов-компонент `PhysicalAddr` и `VirtualAddress`. Например, если `ra` имеет тип `PhysicalAddr`, то `inl ra` будет иметь тип `Addr`. (Имена тегов `inl` и `inr` становятся понятны, если представить их как функции:

```
inl: PhysicalAddr → PhysicalAddr + VirtualAddress;  
inr: VirtualAddress → PhysicalAddr + VirtualAddress;
```

производящие «инъекцию» (вложение) элементов `PhysicalAddr` или `VirtualAddress` в левое или правое подмножество типа-суммы `Addr`. Заметим, однако, что в нашем формальном описании они функциями *не являются*.)

В общем случае, элементы типа $T_1 + T_2$ состоят из элементов T_1 , снабженных тегом `inl`, и элементов T_2 , снабженных тегом `inr`.

Для *использования* типов-сумм мы вводим конструкцию `case`, позволяющую различать, принадлежит ли данное значение левому или правому подмножеству суммы. Например, имя из значения типа `Addr` можно извлечь так:

⁵Эти примеры, подобно большинству случаев реального использования вариантовых типов, используют также *рекурсивные типы* (recursive types) — хвост списка сам является списком, и т. д. К рекурсивным типам мы вернемся в главе 20.

⁶Реализация `fullsimple` на самом деле не поддерживает описанные здесь конструкции с бинарными суммами — вместо этого есть более общая конструкция с вариантами, описанная ниже.

```

getName = λa:Addr .
  case a of
    inl x ⇒ x.firstlast
  | inr y ⇒ y.name;

```

Если в виде параметра a выступает `PhysicalAddr` с тегом `inl`, выражение `case` выполняет первую ветвь и связывает переменную x со значением типа `PhysicalAddr`; тело первой ветви извлекает из x поле `firstlast` и возвращает его. Аналогично, если a есть `VirtualAddress` с тегом `inr`, будет выбрана вторая ветвь и возвращено поле `name` в значении типа `VirtualAddress`. Таким образом, тип всей функции `getName` — `Addr → String`.

Описанные нами конструкции формально представлены на рис. 11.9. Мы добавляем к синтаксису термов левую и правую инъекции, а также конструкцию `case`. К типам мы добавляем конструктор суммы. К правилам вычисления мы добавляем два варианта «бета-редукции» для конструкции `case`: один для случая, когда первый подтерм сведен к значению v_0 с тегом `inl`, а второй — для значения v_0 с тегом `inr`. В обоих случаях мы выбираем соответствующую ветвь и выполняем ее тело, подставляя v_0 вместо связанной переменной. Остальные правила вычисления делают шаг в первом подтерме выражения `case`, либо внутри тегов `inl` или `inr`.

Правила типизации для тегов просты: чтобы показать, что `inl t1` имеет тип-сумму $T_1 + T_2$, достаточно показать, что t_1 принадлежит первому слагаемому типу, T_1 , и сделать то же самое для `inr`. В случае конструкции `case` нужно убедиться в том, что первый подтерм имеет тип $T_1 + T_2$, а также что тела обеих ветвей t_1 и t_2 имеют один и тот же тип T , предполагая, что связанные переменные x_1 и x_2 имеют типы T_1 и T_2 , соответственно; результат всего `case`-выражения имеет тип T . Следуя соглашениям из предыдущих определений, рис. 11.9 не утверждает явно, что область видимости переменных x_1 и x_2 ограничена телами ветвей t_1 и t_2 , но это можно понять по тому, как расширяются контексты в правиле типизации $T\text{-CASE}$.

Упражнение 11.9.1 [★★]: Обратите внимание на сходство между правилом типизации для `case` и правилом для `if` на рис. 8.1: `if` можно рассматривать как своего рода вырожденную форму `case`, в которой в ветви не передается никакая информация. Формализуйте это интуитивное представление, определив `true`, `false` и `if` в виде производных форм на основе типов-сумм и `Unit`.

Суммы и единственность типов

Большинство полезных свойств отношения типизации чистого $\lambda\rightarrow$ (см. §9.3) сохраняются и в системе с типами-суммами. Однако, теряется одно важное свойство: теорема о единственности типов (9.3.3). Сложность возникает из-за конструкций постановки тегов `inl` и `inr`. Например, правило типизации $T\text{-INL}$ говорит, что зная, что t_1 является элементом T_1 , мы можем вывести утверждение, что `inl t1` является элементом $T_1 + T_2$, для *любого* типа T_2 . Например, можно вывести `inl 5 : Nat+Nat` и `inl 5 : Nat+Bool` (и бесконечное число других типов). Несоблюдение единственности типов означает, что невозможно построить алгоритм проверки типов путем «считывания

правил снизу вверх», как мы это делали для всех конструкций, рассмотренных до сих пор. Мы можем выбрать один из нескольких вариантов:

1. Можно усложнить алгоритм проверки типов, чтобы он как-то «догадывался», какое значение T_2 требуется. Мы можем сохранять T_2 неопределенным и попытаться впоследствии определить, какой из типов должен быть на его месте. Такие методы будут подробно рассмотрены, когда мы будем изучать реконструкцию типов (глава 22).
2. Можно расширить язык типов и единообразно представлять *все* типы T_2 одновременно. Этот вариант будет изучен при обсуждении подтипов (глава 15).
3. Можно потребовать, чтобы программист обеспечивал явную *аннотацию* (annotation), указывая, какой тип T_2 имеется в виду. Эта альтернатива самая простая — и, на самом деле, не столь непрактичная, как могло бы показаться, поскольку в полноценных языках явные аннотации часто можно «прицепить» к другим конструкциям и сделать почти невидимыми (мы вернемся к этому вопросу в следующем разделе). Пока что мы выбираем именно этот вариант.

На рис. 11.10 показаны требуемые расширения языка по отношению к рис. 11.9. Вместо простого `inl t` или `inr t` мы пишем `inl t as T` или `inr t as T`, где T обозначает полностью тот тип-сумму, к которому мы намереваемся отнести снабженный тегом элемент. Правила типизации $T\text{-INL}$ и $T\text{-INR}$ используют указанный тип-сумму как тип результата инъекции, предварительно проверяя, что образ терма действительно принадлежит соответствующему подмножеству суммы (чтобы не повторять в правилах $T_1 + T_2$ по многу раз, синтаксические правила позволяют при инъекции указывать любой тип как аннотацию). Правила типизации позаботятся о том, чтобы этот тип всегда был типом-суммой, при условии, что инъекция правильно типизируется.) Синтаксис аннотаций типов специально похож на конструкцию приписывания типов из §11.4: в сущности, эти аннотации можно рассматривать как синтаксически обязательное приписывание типов.

11.10. Варианты

Бинарные типы-суммы являются частным случаем *типов-вариантов* (variant type) с метками, подобно тому, как бинарные типы-произведения являются частным случаем записей с метками полей. Вместо T_1+T_2 мы будем писать $\langle l_1:T_1, l_2:T_2 \rangle$, где l_1 и l_2 — метки полей. Вместо `inl t as T1+T2` будем писать `<L1=t> as <L1:T1, L2:T2>`. А вместо обозначений `inl` и `inr` в вариантах конструкции `case` мы будем использовать те же метки, что и в соответствующем типе-сумме. С такими обобщениями пример функции `getName` из предыдущего раздела будет выглядеть так:

```
Addr = <physical:PhysicalAddr, virtual:VirtualAddress>;
a = <physical=pa> as Addr;
▷ a: Addr
```

```

getName = λa:Addr .
  case a of
    <physical=x> ⇒ x.firstlast
    | <virtual=y> ⇒ y.name;
▷ getName: Addr → String

```

Формальное определение типов-вариантов приводится на рис. 11.11. Обратите внимание, что, как и в записях в §11.8, порядок меток в вариантом типе считается существенным.

Необязательные значения

Одна из весьма полезных идиом с использованием вариантов — *необязательные значения* (optional values). Например, элементом типа

```
OptionalNat = <none:Unit, some:Nat>;
```

может быть либо тривиальное значение `unit` с тегом `none`, либо целое число с тегом `some`. Другими словами, тип `OptionalNat` изоморфен типу `Nat`, к которому добавлено выделенное значение `none`. Тип

```
Table = Nat → OptionalNat;
```

представляет конечные отображения из чисел в числа: областью определения такого отображения служит множество аргументов, на которых результат имеет вид `<some=n>` для некоторого n . Пустая таблица

```

emptyTable = λn:Nat. <none=unit> as OptionalNat;
▷ emptyTable : Table

```

это константная функция, возвращающая `none` в ответ на любое входное значение. Конструктор

```

extendTable =
  λt:Table. λm:Nat. λv:Nat.
  λn:Nat.
    if equal n m then <some=v> as OptionalNat
    else t n;
▷ extendTable: Table → Nat → Nat → Table

```

берет таблицу и добавляет в нее (или изменяет имеющуюся) ячейку, отображающую вход m в выход `<some=v>`. (Функция `equal` определена в решении упражнения 11.11.1 на с. 542.)

Результат, полученный поиском в таблице типа `Table`, можно использовать в выражении `case`. Например, если у нас есть таблица `t`, и мы хотим получить значение, соответствующее числу 5, мы можем написать

```
x = case t(5) of
  <none=u> ⇒ 999
  | <some=v> ⇒ v;
```

выдавая 999 в качестве значения `x` по умолчанию на случай, если `t` не определена для значения 5.

Во многих языках имеется встроенная поддержка необязательных значений. Например, в OCaml есть предопределенный конструктор типов `option`, и

многие функции в типичных программах на OCaml возвращают необязательные значения. Например, `null` в языках вроде C, C++ и Java, в сущности, также представляет собой замаскированное необязательное значение. В этих языках переменная типа `T` (в случае, если `T` — «ссыльочный тип», то есть его значения выделяются в куче) может содержать либо особое значение `null`, либо указатель на значение типа `T`. То есть, тип такой переменной на самом деле есть `Ref(Option(T))`, где `Option(T)=<none:Unit,some:T>`. Конструкция `Ref` подробно обсуждается в главе 13.

Перечисления

Два «вырожденных случая» типов-вариантов достаточно полезны, чтобы упомянуть их отдельно: типы перечислений и варианты с одним полем.

Тип-перечисление (enumeration) — это вариантный тип, в котором с каждым полем ассоциирован тип `Unit`. Например, тип, представляющий рабочие дни недели, можно определить так:

```
Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit,
           thursday:Unit, friday:Unit>;
```

Элементами этого типа являются термы вроде `<monday:Unit> as Weekday`. Действительно, поскольку в типе `Unit` всего один элемент, тип `Weekday` имеет ровно пять значений, однозначно соответствующих дням недели. При помощи конструкции `case` можно определять вычисления с использованием перечислений.

```
nextBusinessDay = λw:Weekday .
  case w of <monday=x>      => <tuesday=unit> as Weekday
            | <tuesday=x>      => <wednesday=unit> as Weekday
            | <wednesday=x>     => <thursday=unit> as Weekday
            | <thursday=x>      => <friday=unit> as Weekday
            | <friday=x>        => <monday=unit> as Weekday;
```

Разумеется, используемый нами конкретный синтаксис плохо приспособлен к написанию и чтению таких программ. Некоторые языки (начиная с Паскаля) обладают особым синтаксисом для объявления перечислений и работы с ними. Другие — например, ML, см. с. 159, — представляют перечисления в виде частного случая вариантных типов.

Варианты с одним полем

Еще один интересный частный случай представляют собой варианты с единственной меткой 1:

```
V = <1:T>;
```

На первый взгляд, такой тип не кажется особенно полезным: в конце концов, все элементы `V` находятся во взаимно однозначном соответствии с элементами поля `T`, поскольку всякий элемент `V` имеет в точности форму `<1=t>` для некоторого `t : T`. Существенно, однако, то, что стандартные операции над `T` *неприменимы* к элементам `V` без распаковки: невозможно случайно перепутать `V` и `T`.

Предположим, например, что мы пишем программу для денежных расчетов в различных валютах. В такой программе могут встретиться функции для перевода долларов в евро. Если и те, и другие представлены как `Float`, эти функции могут выглядеть так:

```
dollars2euros = λd:Float. timesfloat d 1.1325;
> dollars2euros: Float → Float
  euros2dollars = λe:Float. timesfloat e 0.883;
> euros2dollars: Float → Float
```

(где функция `timesfloat: Float -> Float` служит для перемножения чисел с плавающей точкой). Если мы начнем с суммы в долларах,

```
mybankbalance = 39.50;
```

то ее можно перевести в евро и затем обратно в доллары так:

```
euros2dollars (dollars2euros mybankbalance)
> 39.49990125 : Float
```

Все это выглядит вполне разумно. Но с такой же легкостью мы можем производить и совершенно бессмысленные действия. Например, можно перевести сумму на счете в евро дважды:

```
dollars2euros (dollars2euros mybankbalance);
> 50.660971875 : Float
```

Поскольку денежные суммы представлены здесь просто как числа, система типов не может предотвратить бессмыслицу такого рода. Однако, если мы определим доллары и евро как различные вариантные типы (взяв числа с плавающей точкой в качестве внутреннего представления),

```
DollarAmount = <dollars:Float>;
EuroAmount = <euros:Float>;
```

то мы сможем определить безопасные версии функций преобразования валют, в которых на вход непременно должна подаваться сумма в соответствующей валюте:

```
dollars2euros =
  λd:DollarAmount.
    case d of <dollars=x> =>
      <euros = timesfloat x 1.1325> as EuroAmount;
> dollars2euros : DollarAmount → EuroAmount
  euros2dollars =
    λe: EuroAmount.
      case e of <euros=x> =>
        <dollars = timesfloat x 0.883> as DollarAmount;
> euros2dollars : EuroAmount → DollarAmount
```

Теперь процедура проверки типов способна следить за тем, какие валюты используются в вычислениях, и напоминает, как именно следует интерпретировать окончательный результат:

```
mybankbalance = <dollars=39.50> as DollarAmount;
  euros2dollars (dollars2euros mybankbalance);
> <dollars=39.49990125> as DollarAmount : EuroAmount
```

Более того, если написать бессмысленное двойное преобразование, то типы не сойдутся и программа будет отвергнута (как и должно быть):

```
dollars2euros (dollars2euros mybankbalance);
⇒ Error: parameter type mismatch
```

Варианты и типы данных

Вариантный тип $\langle l_i : T_i \rangle_{i \in 1..n}$ приблизительно соответствует типу ML, который определяется так:⁷

```
type T = l1 of T1
      | l2 of T2
      | ...
      | ln of Tn
```

Однако стоит упомянуть несколько отличий.

- Имеется тривиальное, но способное вносить путаницу расхождение между нашими соглашениями по использованию прописных букв и соглашениями, принятыми в OCaml. В OCaml имена типов должны начинаться со строчной буквы, а имена конструкторов (в нашей терминологии, метки вариантов) — с прописной, так что, строго говоря, объявление типа должно записываться так:

```
type t = L1 of t1 | ... | Ln of tn
```

Чтобы не смешивать термы **t** и типы **T**, мы будем в этом обсуждении игнорировать соглашения языка OCaml и продолжим пользоваться нашими собственными.

- Самое интересное различие состоит в том, что OCaml *не требует* аннотации типа, когда элемент типа T_i с помощью конструктора l_i переводится в тип T : можно просто писать $l_i(t)$. OCaml позволяет так делать, сохраняя при этом единственность типов, требуя, чтобы тип T был *объявлен* (be declared) перед использованием. Более того, метки вариантов T не должны использоваться никакими другими типами в той же области видимости. Таким образом, когда процедура проверки типов встречает терм $l_i(t)$, она знает, что правильной аннотацией может быть только T . В сущности, аннотация «спрятана» внутри метки варианта.

Такой прием позволяет избавиться от множества ненужных аннотаций, но вызывает и некоторое недовольство пользователей, поскольку одну метку варианта нельзя использовать в нескольких типах — по крайней мере, в рамках одного модуля. В главе 15 мы рассмотрим другой способ избежать аннотаций, лишенный этого недостатка.

⁷ В этом разделе используется конкретный синтаксис типов данных OCaml, чтобы соответствовать тем главам книги, в которых обсуждается реализация интерпретаторов. Однако такие типы возникли в ранних диалектах ML, и их примерно в той же форме можно найти в Standard ML и в родственных языках вроде Haskell. Типы данных и сопоставление с образцом — вероятно, самые большие преимущества этих языков в ежедневной работе.

3. Еще одна хитрость OCaml заключается в том, если с вариантом связан тип данных `Unit`, то его можно не указывать. В результате перечисления можно определять так:

```
type Weekday = monday | tuesday | wednesday | thursday | friday
```

вместо такой записи:

```
type Weekday = monday of Unit
              | tuesday of Unit
              | wednesday of Unit
              | thursday of Unit
              | friday of Unit
```

Кроме того, метка варианта `monday` сама по себе (а не метка `monday`, примененная к вырожденному значению `unit`) считается значением типа `Weekday`.

4. Наконец, типы данных OCaml увязывают вариантные типы с несколькими дополнительными конструкциями, которые мы исследуем по отдельности в последующих главах.

- Определение типа данных может быть *рекурсивным* (recursive), т. е. имя определяемого типа может встречаться в теле определения. Например, при стандартном определении списка натуральных чисел значение с тегом `cons` — это пара, второй элемент которой имеет тип `NatList`.

```
type NatList = nil
             | cons of Nat * NatList
```

- Тип OCaml может быть *параметризован* (parametrized) типовой переменной, как, например, в обобщенном определении типа списка `List`:

```
type 'a List = nil
             | cons of 'a * 'a List
```

С точки зрения теории типов можно рассматривать `List` как особыго рода функцию, которая называется *оператором над типами* (type operator), и которая переводит каждый тип-аргумент `'a` в конкретный тип данных: `Nat` в `NatList`, и т. д. Типовые операторы обсуждаются в главе 29.

Варианты как непересекающиеся объединения

Типы-суммы и вариантные типы иногда называют *непересекающимися объединениями* (disjoint unions). Тип $T_1 + T_2$ есть «объединение» T_1 и T_2 в том смысле, что элементы этого типа включают в себя все элементы T_1 и T_2 . Это непересекающееся объединение, поскольку множества элементов T_1 и T_2 перед собственно объединением помечены тегами, соответственно, `inl` и `inr`. Таким образом, всегда можно легко узнать, происходит ли тот или иной элемент из

T_1 или из T_2 . Выражение *тип-объединение* (union type) также применяется к непомеченным (пересекающимся) типам объединений, которые описаны в §15.7.

Тип Dynamic

Даже в языках со статической типизацией бывает необходимо иметь дело с данными, тип которых невозможно определить на этапе компиляции. В частности, такое происходит, когда данные существуют на нескольких различных машинах или в течение нескольких запусков программы — например, когда эти данные хранятся в базе данных или передаются по компьютерной сети. Чтобы справляться с такими ситуациями безопасным образом, во многих языках присутствуют средства, позволяющие определить тип значений во время работы программы.

Один из хороших способов добиться этого заключается в добавлении типа `Dynamic`, значения которого представляют собой пары из значений `v` и меток типа `T`, где `v` имеет тип `T`. Элементы типа `Dynamic` строятся при помощи особой конструкции присваивания меток, а анализируются с помощью безопасной конструкции определения типа `typecase`. В сущности, можно рассматривать `Dynamic` как непересекающееся объединение бесконечного размера, в котором метками служат типы. См. работы Гордона (Gordon, circa 1980), Майкрофта (Mycroft, 1983), Абади, Карделли, Пирса и Плоткина (Abadi, Cardelli, Pierce, and Plotkin, 1991b), Леруа и Мони (Leroy and Mauny, 1991), Абади, Карделли, Пирса и Реми (Abadi, Cardelli, Pierce, and Rémy, 1995), а также Хенглейна (Henglein, 1994).

11.11. Рекурсия общего вида

Еще одна возможность, доступная во многих языках программирования — определение рекурсивных функций. Мы уже видели (глава 5, с. 84), что в бестиповом лямбда-исчислении такие функции можно определять с помощью комбинатора `fix`.

В типизированном языке рекурсивные функции можно определять аналогично. Вот, например, функция `iseven`, которая возвращает `true`, будучи вызвана с четным аргументом, и `false` в противном случае:

```
ff = λie:Nat → Bool.
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else ie (pred (pred x));

▷ ff : (Nat → Bool) → Nat → Bool

iseven = fix ff;

▷ iseven : Nat → Bool
```

```
iseven 7;
> false : Bool
```

Интуитивное представление здесь таково: функция высшего порядка `ff`, передаваемая аргументом в `fix`, является *генератором* (generator) для функции `iseven`: если `ff` применяется к функции `ie`, поведение которой приближенно соответствует требуемому поведению `iseven` вплоть до некоторого числа n (то есть, к функции, возвращающей правильные результаты, когда ее аргумент меньше или равен n), то она возвращает более точное приближение к `iseven` — функцию, выдающую правильные результаты вплоть до аргумента $n + 2$. Применение `fix` к этому генератору возвращает его неподвижную точку — функцию, возвращающую правильный результат для любого входного значения n .

Есть, однако, одно важное отличие от бестипового языка: сама функция `fix` не может быть определена в рамках простого типизированного лямбда-исчисления. Более того, в главе 12 мы покажем, что *никакое* выражение, способное привести к незавершающемуся вычислению, не может быть типизировано исключительно средствами простых типов.⁸ Поэтому вместо того, чтобы определять `fix` как терм нашего языка, мы просто добавляем его в качестве нового примитива, вместе с правилами вычисления, имитирующими поведение бестипового комбинатора `fix`, и правилом типизации, отражающим его реальное использование. Эти правила изображены на рис. 11.12. (Сокращенная форма `letrec` будет описана далее.)

Простое типизированное лямбда-исчисление с добавлением чисел и комбинатора `fix` давно уже служит любимым объектом исследований для специалистов по языкам программирования, поскольку это простейший язык, в котором возникает множество тонких семантических явлений, например, *полная абстракция* (full abstraction) (Plotkin, 1977, Hyland and Ong, 2000, Abramsky, Jagadeesan, and Malacaria, 2000). Часто это исчисление называют *PCF* («Programming language for Computable Functions», «Язык программирования вычислимых функций»).

Упражнение 11.11.1 [★★]: Определите с помощью `fix` функции `equal`, `plus`, `times` и `factorial`.

Как правило, конструкция `fix` используется для построения функций (в качестве неподвижных точек функций, переводящих функции в функции), однако имеет смысл заметить, что тип T в правиле T -`Fix` не обязательно должен быть функциональным. Иногда эта дополнительная гибкость оказывается полезной. Например, она позволяет определить *запись*, состоящую из взаимно рекурсивных функций, как неподвижную точку функции, работающей с записями (содержащими функции). Следующая реализация `iseven` использует вспомогательную функцию `isodd`; эти две функции определены как поля в записи, причем определение этой записи абстрагируется от записи `ieio`, и ее компоненты используются для рекурсивных вызовов из тел полей `iseven` и `isodd`.

⁸ В дальнейшем, в главе 13 и главе 20, мы введем в систему простых типов некоторые расширения, возвращающие способность определить `fix` средствами самой системы.

```

ff = λieio:{iseven:Nat → Bool, isodd:Nat → Bool}.
  {iseven = λx:Nat.
    if iszero x then true
    else ieio.isodd (pred x),
  isodd = λx:Nat.
    if iszero x then false
    else ieio.iseven (pred x)};
▷ ff : {iseven:Nat → Bool, isodd:Nat → Bool} →
      {iseven:Nat → Bool, isodd:Nat → Bool}

```

Неподвижная точка функции `ff` — это запись из двух функций,

```

r = fix ff;
▷ r : {iseven:Nat → Bool, isodd:Nat → Bool}

```

и обращение к первой из них дает нам саму функцию `iseven`:

```

iseven = r.iseven;
▷ iseven : Nat → Bool
iseven 7;
▷ false : Bool

```

Возможность создания неподвижной точки функции типа $T \rightarrow T$ для любого T приводит к некоторым неожиданным последствиям. В частности, оказывается, что *каждый* тип имеет хотя бы один терм этого типа. Чтобы показать это, заметим, что для каждого типа T мы можем определить функцию `divergeT`:

```

divergeT = λ_:Unit. fix (λx:T.x);
▷ divergeT : Unit → T

```

Каждый раз, когда `divergeT` применяется к аргументу `unit`, мы получаем бесконечную последовательность вычислений, в которой снова и снова применяется правило E-FixБЕТА, всегда давая один и тот же терм. Получается, что для каждого типа T терм `divergeT unit` служит *неопределенным элементом* (*undefined element*).

Последнее усовершенствование, которое мы рассмотрим — более удобный конкретный синтаксис для часто встречающейся ситуации, в которой требуетсѧ связать переменную с результатом рекурсивного определения. В большинстве высокоуровневых языков первое приведенное нами определение `iseven` было бы записано примерно так:

```

letrec iseven : Nat → Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
iseven 7;
▷ false : Bool

```

Конструкцию рекурсивного связывания `letrec` несложно определить в качестве производной формы:

$$\text{letrec } x:T_1=t_1 \text{ in } t_2 \stackrel{\text{def}}{=} \text{let } x = \text{fix } (\lambda x:T_1.t_1) \text{ in } t_2$$

Упражнение 11.11.2 [★]: Перепишите свои определения функций `plus`, `times` и `factorial` из упражнения 11.11.1 через `letrec` вместо `fix`.

Более подробную информацию об операторах неподвижной точки можно найти у Клопа (Klop, 1980) и Винскеля (Winskel, 1993).

11.12. Списки

Рассмотренные нами конструкции можно разделить на *базовые типы* (base types), такие как `Bool` и `Unit`, и *конструкторы типов* (type constructors), например, \rightarrow и \times , которые строят новые типы на основе старых. Еще один полезный конструктор — `List`. Для каждого типа `T`, тип `List T` описывает списки конечной длины, состоящие из элементов типа `T`.

На рис. 11.13 показаны синтаксис, семантика и правила типизации для списков. С точностью до синтаксических расхождений (`List T` вместо `T list` и т. п.), а также того, что в нашем варианте все синтаксические формы требуют явной аннотации типов, наши списки, по существу, совпадают со списками, имеющимися в ML и других функциональных языках. Пустой список (с элементами типа `T`) записывается в виде `nil[T]`. Список, порождаемый при добавлении нового элемента `t1` (типа `T`) в начало списка `t2`, записывается в виде `cons[T] t1 t2`. Голова и хвост списка `t` записываются как `head[T] t` и `tail[T] t`. Булевский предикат `isnil[T]` `t` выдает `true` тогда и только тогда, когда список `t` пуст.⁹

Упражнение 11.12.1 [★★★]: Покажите, что в простом типизированном лямбда-исчислении с булевскими значениями и списками выполняются теоремы о продвижении и сохранении.

Упражнение 11.12.2 [★★]: Представленный здесь синтаксис списков включает множество аннотаций, которые на самом деле не нужны, поскольку правила типизации легко могли бы вывести нужный тип из контекста. (Эти аннотации предназначены для облегчения сравнения с кодированием списков в §23.4). Можно ли избавиться от всех аннотаций?

⁹Мы приняли «представление списков через `head/tail/isnil`» ради простоты. С точки зрения проектирования языков, вероятно, правильнее было бы рассматривать списки как тип данных и обращаться к их компонентам через выражения `case`, поскольку при этом большее число программистских ошибок становится ошибками типизации.

$\rightarrow +$ Расширяет $\lambda \rightarrow$ (9.1)

Новые синтаксические формы

$t ::= \dots$	<i>термы:</i>
inl t	постановка тега (левого)
inr t	постановка тега (правого)
case t of inl x=>t inr x=>t	
<i>case</i>	

$v ::= \dots$	<i>значения:</i>
inl v	значение с тегом (левым)
inr v	значение с тегом (правым)

$T ::= \dots$	<i>типы:</i>
T+T	<i>тип-сумма</i>

Новые правила вычисления $t \rightarrow t'$

$$\begin{aligned} &\text{case (inl } v_0) \\ &\text{of inl } x_1=>t_1 \mid \text{inr } x_2=>t_2 \\ &\rightarrow [x_1 \mapsto v_0]t_1 \end{aligned}$$

(E-CASEINL)

$$\begin{aligned} &\text{case (inr } v_0) \\ &\text{of inl } x_1=>t_1 \mid \text{inr } x_2=>t_2 \\ &\rightarrow [x_2 \mapsto v_0]t_2 \end{aligned}$$

(E-CASEINR)

$$\frac{t_0 \rightarrow t'_0}{\text{case } t_0 \text{ of inl } x_1=>t_1 \mid \text{inr } x_2=>t_2 \rightarrow \text{case } t'_0 \text{ of inl } x_1=>t_1 \mid \text{inr } x_2=>t_2} \quad (\text{E-CASE})$$

$$\frac{t_1 \rightarrow t'_1}{\text{inl } t_1 \rightarrow \text{inl } t'_1} \quad (\text{E-INL})$$

$$\frac{t_1 \rightarrow t'_1}{\text{inr } t_1 \rightarrow \text{inr } t'_1} \quad (\text{E-INR})$$

Новые правила типизации $\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 : T_1 + T_2} \quad (\text{T-INL})$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \text{inr } t_1 : T_1 + T_2} \quad (\text{T-INR})$$

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \quad \Gamma, x_1:T_1 \vdash t_1 : T \quad \Gamma, x_2:T_2 \vdash t_2 : T}{\Gamma \vdash \text{case } t_0 \text{ of inl } x_1=>t_1 \mid \text{inr } x_2=>t_2 : T} \quad (\text{T-CASE})$$

Рис. 11.9. Типы-суммы

$\rightarrow +$ *Расширяет λ_{\rightarrow} (11.9)***Новые синтаксические формы** $t ::= \dots$ *термы:*inl t **as T** постановка тега

(левого)

inr t **as T** постановка тега

(правого)

 $v ::= \dots$ *значения:*inl v **as T** значение с

тегом (левым)

inr v **as T** значение с

тегом (правым)

$$\begin{aligned} &\text{case (inr } v_0 \text{ as } T_0 \text{)} \\ &\text{of inl } x_1 = t_1 \mid \text{inr } x_2 = t_2 \\ &\quad \rightarrow [x_2 \mapsto v_0]t_2 \end{aligned}$$
Новые правила вычисления $t \rightarrow t'$

(E-CASEINR)

$$\frac{}{\text{inl } t_1 \text{ as } T_2 \rightarrow \text{inl } t'_1 \text{ as } T_2} \quad t_1 \rightarrow t'_1$$

(E-INL)

$$\begin{aligned} &\text{case (inl } v_0 \text{ as } T_0 \text{)} \\ &\text{of inl } x_1 = t_1 \mid \text{inr } x_2 = t_2 \\ &\quad \rightarrow [x_1 \mapsto v_0]t_1 \end{aligned}$$

$$\frac{}{\text{inr } t_1 \text{ as } T_2 \rightarrow \text{inr } t'_1 \text{ as } T_2} \quad t_1 \rightarrow t'_1$$

(E-INR)

Новые правила типизации

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 : T_1 + T_2} \quad (\text{T-INL})$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \text{inr } t_1 \text{ as } T_1 + T_2 : T_1 + T_2} \quad (\text{T-INR})$$

Рис. 11.10. Суммы (с единственностью типизации)

$\rightarrow \langle \rangle$ Расширяет λ_{\rightarrow} (9.1)

Новые синтаксические формы

$t ::= \dots$ термы:
 $\langle l=t \rangle \text{ as } T$ постановка тега
 $\text{case } t \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n}$
 case

$v ::= \dots$ значения:
 $\langle l=v \rangle \text{ as } T$ значение с тегом
 $T ::= \dots$ типы:
 $\langle l_i:T_i^{i \in 1..n} \rangle$ тип вариантов

Новые правила вычисления

 $t \rightarrow t'$

$\text{case } (\langle l_j=v_j \rangle \text{ as } T) \text{ of}$
 $\quad \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n}$
 $\quad \rightarrow [x_j \mapsto v_j]t_j$

(E-CASEVARIANT)

$$\frac{t_0 \rightarrow t'_0}{\text{case } t_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n}}$$

$$\rightarrow \text{case } t'_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n}$$

(E-CASE)

$$\frac{t_i \rightarrow t'_i}{\langle l_i=t_i \rangle \text{ as } T \rightarrow \langle l_i=t'_i \rangle \text{ as } T}$$

(E-VARIANT)

Новые правила типизации

 $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \langle l_j=t_j \rangle \text{ as } \langle l_i:T_i^{i \in 1..n} \rangle : \langle l_i:T_i^{i \in 1..n} \rangle}$$

(T-VARIANT)

$$\frac{\Gamma \vdash t_0 : \langle l_i:T_i^{i \in 1..n} \rangle \text{ для каждого } i: \Gamma, x_i:T_i \vdash t_i:T}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n} : T}$$

(T-CASE)

Рис. 11.11. Варианты

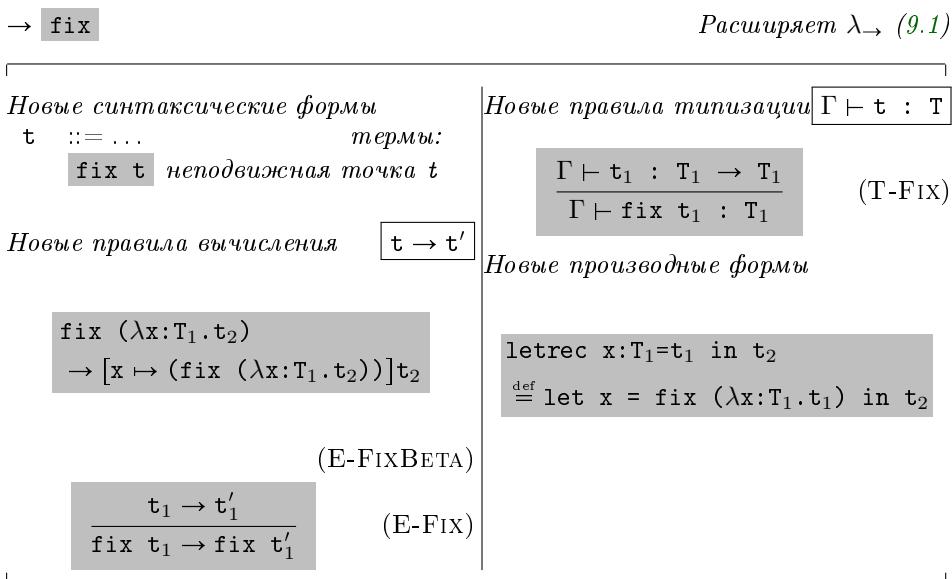


Рис. 11.12. Рекурсия общего вида

→ B List

Расширяет λ_{\rightarrow} (9.1) с булевскими значениями (8.1)

Новые синтаксические формы

$t ::= \dots$	термы:
$\text{nil}[T]$	пустой список
$\text{cons}[T] \ t \ t$	конструктор списка
$\text{isnil}[T] \ t$	проверка на пустой список
$\text{head}[T] \ t$	голова списка
$\text{tail}[T] \ t$	хвост списка

$v ::= \dots$	значения:
$\text{nil}[T]$	пустой список
$\text{cons}[T] \ v \ v$	конструктор списка
$\text{isnil}[T]$	

$T ::= \dots$	типы:
$\text{List } T$	тип списков

Новые правила вычисления

$$t \rightarrow t'$$

$$\frac{t_1 \rightarrow t'_1}{\text{cons}[T] \ t_1 \ t_2 \rightarrow \text{cons}[T] \ t'_1 \ t_2} \quad (\text{E-CONS1})$$

$$\frac{t_2 \rightarrow t'_2}{\text{cons}[T] \ v_1 \ t_2 \rightarrow \text{cons}[T] \ v_1 \ t'_2} \quad (\text{E-CONS2})$$

$$\text{isnil}[S] \ (\text{nil}[T]) \rightarrow \text{true} \quad (\text{E-ISNILNIL})$$

$$\text{isnil}[S] \ (\text{cons}[T] \ v_1 \ v_2) \rightarrow \text{false} \quad (\text{E-ISNILCONS})$$

$$\frac{t_1 \rightarrow t'_1}{\text{isnil}[T] \ t_1 \rightarrow \text{isnil}[T] \ t'_1} \quad (\text{E-ISNIL})$$

$$\text{head}[S] \ (\text{cons}[T] \ v_1 \ v_2) \rightarrow v_1 \quad (\text{E-HEADCONS})$$

$$\frac{t_1 \rightarrow t'_1}{\text{head}[T] \ t_1 \rightarrow \text{head}[T] \ t'_1} \quad (\text{E-HEAD})$$

$$\text{tail}[S] \ (\text{cons}[T] \ v_1 \ v_2) \rightarrow v_2 \quad (\text{E-TAILCONS})$$

$$\frac{t_1 \rightarrow t'_1}{\text{tail}[T] \ t_1 \rightarrow \text{tail}[T] \ t'_1} \quad (\text{E-TAIL})$$

Новые правила типизации $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \text{nil}[T_1] : \text{List } T_1 \quad (\text{T-NIL})$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : \text{List } T_1}{\Gamma \vdash \text{cons}[T_1] \ t_1 \ t_2 : \text{List } T_1} \quad (\text{T-CONS})$$

$$\frac{\Gamma \vdash t_1 : \text{List } T_{11}}{\Gamma \vdash \text{isnil}[T_{11}] \ t_1 : \text{Bool}} \quad (\text{T-ISNIL})$$

$$\frac{\Gamma \vdash t_1 : \text{List } T_{11}}{\Gamma \vdash \text{head}[T_{11}] \ t_1 : T_{11}} \quad (\text{T-HEAD})$$

$$\frac{\Gamma \vdash t_1 : \text{List } T_{11}}{\Gamma \vdash \text{tail}[T_{11}] \ t_1 : \text{List } T_{11}} \quad (\text{T-TAIL})$$

Рис. 11.13. Списки

Глава 12

Нормализация

В этой главе мы рассматриваем еще одно фундаментальное теоретическое свойство чистого простого типизированного лямбда-исчисления: то, что вычисление правильно типизированной программы гарантированно останавливается за конечное число шагов — т. е., каждый правильно типизированный терм *нормализуем* (normalizable).

Полноценные языки программирования не обладают свойством нормализации, в отличие от других свойств типовой безопасности, рассмотренных нами ранее. Это потому, что такие языки почти всегда расширяют простое типизированное лямбда-исчисление конструкциями вроде рекурсии общего вида (§11.11) или рекурсивных типов (гл. 20), с помощью которых можно создавать зацикливающиеся программы. Однако понятие нормализации снова возникнет, уже на уровне *типов*, когда в §30.3 мы будем обсуждать метатеорию Системы F_ω : в этой системе язык типов, по существу, дублирует простое типизированное лямбда-исчисление, и завершимость алгоритма проверки типов полагается на то, что операция «нормализации» на выражениях, составленных из типов, гарантированно завершается.

Еще один повод изучать доказательства нормализации заключается в том, что это едва ли не самые изящные и восхитительные математические конструкции во всей литературе по теории типов. Зачастую (как в данном случае), они используют базовый метод доказательств через *логические отношения* (logical relations).

Некоторым читателям будет проще пропустить данную главу при первом чтении; в последующих главах никаких сложностей из-за этого не возникнет. (Полная таблица зависимостей между главами приведена на с. 14.)

12.1. Нормализация для простых типов

Рассматриваемый здесь язык — простое типизированное лямбда-исчисление с единственным базовым типом A . Доказательство нормализации

В этой главе изучается язык простого типизированного лямбда-исчисления (рис. 9.1) с одним базовым типом A (11.1).

для этого исчисления не вполне тривиально, поскольку при каждой редукции терма в подтермах могут создаваться множественные редексы.

Упражнение 12.1.1 [★]: Где именно терпит неудачу попытка доказать нормализацию при помощи простой индукции по размеру правильно типизированного терма?

Основная сложность здесь (как и во многих других доказательствах по индукции) в том, чтобы найти достаточно сильное предположение индукции. С этой целью определим для каждого типа T множество R_T замкнутых термов типа T . Рассматривая эти множества как предикаты, введем обозначение $R_T(t)$ для $t \in R_T$.¹

Определение 12.1.2

- $R_A(t)$ тогда и только тогда, когда t завершается.
- $R_{T_1 \rightarrow T_2}(t)$ тогда и только тогда, когда t завершается и для всякого s такого, что $R_{T_1}(s)$, выполняется $R_{T_2}(t s)$.

Это определение дает нам требуемое усиление предположения индукции. Наша главная цель — доказать, что завершаются все *программы*, т. е. все замкнутые термы базового типа. Однако замкнутые термы базового типа могут содержать подтермы функционального типа, так что и об этих термах нужно кое-что знать. Более того, недостаточно знать, что эти подтермы завершаются, поскольку при применении нормализованной функции к нормализованному аргументу происходит подстановка, и эта подстановка может привести к новым шагам вычисления. Таким образом, для термов функционального типа необходимо более сильное условие: они не только должны завершаться сами, но и, будучи примененными к завершающимся аргументам, также должны давать завершающиеся результаты.

Форма определения 12.1.2 характерна для метода доказательств через *логические отношения* (logical relations). (Поскольку сейчас мы работаем только с одноместными отношениями, правильнее было бы сказать *логические предикаты*, logical predicates.) Если мы хотим доказать, что некоторое свойство P выполняется для всех замкнутых термов типа A , мы с помощью индукции на типах доказываем, что все термы типа A *обладают* свойством P , что все термы типа $A \rightarrow A$ *сохраняют* свойство P , что все термы типа $(A \rightarrow A) \rightarrow (A \rightarrow A)$ *сохраняют* свойство *сохранения* свойства P , и так далее. Мы добиваемся этого, определяя семейство предикатов, проиндексированных типами. Для базового типа A этот предикат — просто P . Для функциональных типов предикат утверждает, что функция должна переводить значения, удовлетворяющие предикату для входного типа, в значения, удовлетворяющие предикату для типа-результата.

¹ Иногда множества R_T называют *насыщенными множествами* (saturated sets) или *множествами кандидатов на редуцируемость* (reducibility candidates).

При помощи этого определения мы проводим доказательство нормализации в два шага. Сначала убедимся в том, что каждый элемент каждого множества R_T нормализуем. Затем покажем, что каждый правильно типизированный терм типа T является элементом R_T .

Первый шаг непосредственно следует из определения R_T :

Лемма 12.1.3 *Если верно $R_T(t)$, то t завершается.*

Второй шаг разбит на две леммы. Заметим сначала, что принадлежность множеству R_T сохраняется при вычислении.

Лемма 12.1.4 *Если $t : T$ и $t \rightarrow t'$, то $R_T(t)$ тогда и только тогда, когда $R_T(t')$.*

Доказательство: Индукция по структуре типа T . Во-первых, очевидно, что t завершается тогда и только тогда, когда завершается t' . Если $T = A$, доказывать больше нечего. С другой стороны, предположим, что $T = T_1 \rightarrow T_2$ для некоторых T_1 и T_2 . Для направления «только тогда» (\Rightarrow) предположим, что $R_T(t)$, и что $R_{T_1}(s)$ для некоторого произвольного $s : T_1$. По определению, имеем $R_{T_2}(t s)$. Однако $t s \rightarrow t' s$, и отсюда, по предположению индукции для типа T_2 , получаем $R_{T_2}(t' s)$. Поскольку это верно для любого s , определение R_T дает нам $R_T(t')$. Рассуждение в направлении «тогда» (\Leftarrow) проводится аналогично.

Теперь нам нужно показать, что каждый терм типа T принадлежит множеству R_T . Здесь индукция будет проводиться по деревьям вывода типов (было бы странно, если бы какое-либо доказательство, связанное с правильно типизированными термами, не содержало индукции по деревьям вывода типов!). Единственная сложность здесь заключается в обработке случая с λ -абстракцией. Поскольку мы проводим индукцию, доказательство того, что терм $\lambda x:T_1. t_2$ принадлежит множеству $R_{T_1} \rightarrow T_2$, должно использовать предположение индукции, чтобы показать, что t_2 принадлежит множеству R_{T_2} . Однако R_{T_2} определяется как множество замкнутых термов, в то время как переменная x может быть свободна в t_2 , так что этот способ доказательства не работает.

Эта проблема решается с помощью стандартного приема: нужно найти подходящее обобщение предположения индукции. Вместо того, чтобы доказывать утверждение о замкнутых термах, докажем его обобщенную форму, включающую все замкнутые экземпляры открытого терма t .

Лемма 12.1.5 *Если $x_1:T_1, \dots, x_n:T_n \vdash t : T$, а v_1, \dots, v_n — замкнутые значения типов T_1, \dots, T_n с $R_{T_i}(v_i)$ для каждого i , то $R_T([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] t)$.*

Доказательство: Индукция по дереву вывода утверждения $x_1:T_1, \dots, x_n:T_n \vdash t : T$. (Наиболее интересен вариант с абстракцией.)

Вариант T-VAR: $t = x_i$ $T = T_i$

Утверждение леммы следует немедленно.

Вариант T-ABS: $t = \lambda x:S_1. s_2$ $x_1:T_1, \dots, x_n:T_n, x:S_1 \vdash s_2 : S_2$
 $T = S_1 \rightarrow S_2$

Очевидно, $[x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] t$ дает при вычислении значение, поскольку уже является значением. Остается показать, что $R_{S_2}(([x_1 \mapsto$

$v_1] \cdots [x_n \mapsto v_n] t) s)$ для всякого терма $s : S_1$ такого, что $R_{S_1}(s)$. Предположим, что s — такой терм. Согласно лемме 12.1.3, имеем $s \rightarrow^* v$ для некоторого v . Согласно лемме 12.1.4, $R_{S_1}(v)$. По предположению индукции, $R_{S_2}([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] [x \mapsto v] s_2)$. Однако

$$\begin{aligned} & (\lambda x:S_1. [x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] s_2) \ s \\ \rightarrow^* & [x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] [x \mapsto v] s_2, \end{aligned}$$

а отсюда, по лемме 12.1.4,

$$R_{S_2}((\lambda x:S_1. [x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] s_2) \ s),$$

то есть, $R_{S_2}([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] (\lambda x:S_1. s_2) \ s)$. Поскольку s был выбран произвольно, по определению $R_{S_1 \rightarrow S_2}$, имеем

$$R_{S_1 \rightarrow S_2}([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] (\lambda x:S_1. s_2)).$$

Вариант Т-APP: $t = t_1 \ t_2$

$$\begin{aligned} & x_1:T_1, \dots x_n:T_n \vdash t_1 : T_{11} \rightarrow T_{12} \\ & x_1:T_1, \dots x_n:T_n \vdash t_2 : T_{11} \\ & T = T_{12} \end{aligned}$$

Предположение индукции дает нам $R_{T_{11} \rightarrow T_{12}}([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] t_1)$ и $R_{T_{11}}([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] t_2)$. По определению $R_{T_{11} \rightarrow T_{12}}$,

$$R_{T_{12}}(([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] t_1) \ ([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] t_2))$$

m. e.,

$$R_{T_{12}}([x_1 \mapsto v_1] \cdots [x_n \mapsto v_n] (t_1 \ t_2))$$

Теперь свойство нормализации является простым следствием, если считать, что терм t в лемме 12.1.5 замкнут, а затем использовать то обстоятельство, что все элементы R_T нормализующие для всякого T .

Теорема 12.1.6 [Нормализация]: Если $\vdash t : T$, то t нормализуем.

Доказательство: $R_T(t)$ по лемме 12.1.5; следовательно, t нормализуем по лемме 12.1.3.

Упражнение 12.1.7 [Рекомендуется, ★★★]: С помощью методов этой главы покажите, что простое типизированное лямбда-исчисление сохраняет свойство нормализации, будучи расширено булевскими значениями (рис. 3.1) и типами-произведениями (рис. 11.5).

12.2. Дополнительные замечания

В теоретической литературе свойство нормализации чаще всего формулируется как *строгая нормализация* (strong normalization) для исчислений с полной (недетерминистской) бета-редукцией. Стандартный метод доказательства был изобретен Тейтом (1967); Жирар обобщил его на Систему F (1972,

1989, ср. гл. 23); затем Тейт упростил доказательство (Tait, 1975). В этой книге мы адаптируем метод Тейта для вызова по значению, как это сделал Мартин Хоффман (частное сообщение). Среди классических справочников по методу логических отношений — работы Говарда (Howard, 1973), Тейта (Tait, 1967), Фридмана (Friedman, 1975), Плоткина (Plotkin, 1973, 1980) и Стэтмана (Statman, 1982, 1985a,b). Этот метод также описывается во многих работах по семантике, например, в книгах Митчелла (Mitchell, 1996) и Гантера (Gunter, 1992).

Доказательство строгой нормализации по Тейту в точности соответствует алгоритму вычисления термов с простой типизацией, известному как *нормализация вычислением* (normalization by evaluation) или *частичное вычисление, управляемое типами* (type-directed partial evaluation) (Berger, 1993; Danvy, 1998); см. также работы Бергера и Швихтенберга (Berger and Schwichtenberg, 1991), Филински (Filinski, 1999, 2001) и Рейнольдса (Reynolds, 1998a).

Глава 13

Ссылки

До сих пор мы рассматривали различные *чистые* (pure) языковые конструкции, в том числе функциональную абстракцию, базовые типы, такие как числа и булевские значения, и структурированные типы, такие как записи и вариантные типы. Эти конструкции составляют основу большинства языков программирования, включая чисто функциональные, такие, как Haskell, «функциональные по большей части», такие, как ML, императивные (как C) и объектно-ориентированные (как Java).

В большинстве практических языков программирования имеются также различные *нечистые* (impure) конструкции, которые невозможно описать в рамках простой семантической модели, использовавшейся нами до сих пор. В частности, помимо порождения результатов, вычисление термов в этих языках может приводить к присваиванию значений изменяемым переменным (адресуемым ячейкам, массивам, изменяемым полям записей, и т. п.); вводу и выводу в файлы, на дисплеи, по сетевым соединениям; нелокальной передаче управления с помощью исключений, переходов или продолжений; синхронизации и обмену информацией между процессами, и так далее. В литературе по языкам программирования такие «побочные эффекты» вычисления обычно называют *вычислительными эффектами* (computational effects).

В этой главе мы увидим, как одна из разновидностей вычислительных эффектов — изменяемые ссылки — может быть добавлена в изучаемые нами исчисления. Основное добавление будет заключаться в явных операциях с *памятью* (store) (или *кучей*, heap). Это расширение нетрудно определить; самое интересное начнется, когда возникнет необходимость уточнить теорему о сохранении типов (13.5.3). Еще один вид эффектов, исключения и нелокальную передачу управления, мы рассмотрим в главе 14.

В этой главе рассматривается простое типизированное лямбда-исчисление с типом `Unit` и ссылками (рис. 13.1). Соответствующая реализация на OCaml называется `fullref`.

13.1. Введение

Почти все языки программирования¹ обладают в том или ином виде операцией *присваивания* (assignment), которая изменяет содержимое заранее выделенного фрагмента памяти. В некоторых языках, в частности, в ML и родственных ему, механизмы связывания имен и присваивания разделены. Можно иметь переменную *x*, значением которой является число 5, а можно иметь переменную *y*, значением которой является *ссылка* (referencе) (или *указатель*, pointer) на изменяемую ячейку с текущим содержимым 5, и программисту видно это различие. Можно сложить *x* с другим числом, но присвоить ему новое значение нельзя. Переменную *y* можно напрямую использовать для присваивания нового значения ячейке, на которую она указывает (это записывается в виде *y := 84*), но ее нельзя просто употребить в качестве аргумента *plus*. Для этой цели требуется явно *разыменовать* (dereference) ссылку, написав *!y* для обращения к ее текущему содержимому. В большинстве других языков — в частности, во всех членах семейства C, включая Java, — каждое имя переменной обозначает изменяемую ячейку, и операция разыменования переменной с целью получения ее текущего значения производится неявно.²

С точки зрения формального исследования полезно разделять эти механизмы;³ наше описание в этой главе будет близко следовать модели ML. Уроки этого описания легко можно перенести на языки семейства C, если забыть некоторые различия и сделать кое-какие операции, вроде обращения по ссылке, неявными.

Основные понятия

Основные операции над ссылками — *выделение памяти* (allocation), *разыменование* (dereferencing) и *присваивание* (assignment). Для выделения памяти для ссылки используется оператор *ref*, которому в качестве аргумента дается начальное значение новой ячейки.

```
r = ref 5;
▷ r : Ref Nat
```

Ответ программы проверки типов указывает, что значением *r* является ссылка на ячейку, которая всегда будет содержать число. Чтобы прочитать текущее значение этой ячейки, мы используем оператор разыменования *!* («восклицательный знак»).

¹Даже «чисто функциональные» языки, такие как Haskell, через расширения вроде монад.

²Строго говоря, правильнее рассматривать большинство переменных типа T в C или Java как *указатели* на ячейки, содержащие значения типа Option(T), поскольку содержимым переменной может быть либо обычное значение, либо особое значение null.

³Можно также привести доводы в пользу того, что такое разделение благотворно и с точки зрения проектирования языков. Когда использование изменяемых ячеек памяти оказывается явной операцией, а не умолчанием, это стимулирует по большей части функциональный стиль программирования, в котором ссылки используются относительно редко; такая практика значительно облегчает написание, поддержку и анализ программ, особенно если в языке есть конструкции вроде параллелизма.

```

!r;
> 5 : Nat

```

Чтобы изменить значение, содержащееся в ячейке, используется оператор присваивания.

```

r := 7;
> unit : Unit

```

(Результатом присваивания является вырожденное значение `unit`; см. §11.2.) Если мы снова разыменуем `r`, мы получим новое значение.

```

!r;
> 7 : Nat

```

Побочные эффекты и последовательность действий

То, что результатом выражения присваивания является вырожденное значение `unit`, хорошо сочетается с нотацией *последовательного исполнения* (sequencing), определенной в §11.3, которая позволяет писать

```

(r := succ(!r); !r);
> 8 : Nat

```

вместо эквивалентного, но более громоздкого

```

(λ_:Unit. !r) (r := succ(!r));
> 9 : Nat

```

для того, чтобы выполнить по порядку два выражения и вернуть значение второго. Требование, гласящее, что тип первого выражения должен быть `Unit`, помогает программе проверки типов выловить некоторые глупые ошибки: первое значение разрешается отбросить, только если оно действительно гарантированно вырождено.

Заметим, что, если второе выражение также является присваиванием, то тип всей последовательности снова будет `Unit`. Поэтому мы имеем право опять использовать его слева от точки с запятой и построить еще более длинную цепочку присваиваний:

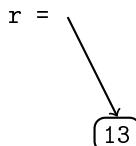
```

(r := succ(!r); r := succ(!r); r := succ(!r); r := succ(!r); !r);
> 13 : Nat

```

Ссылки и псевдонимы

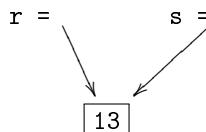
Необходимо помнить о различии между *ссылкой* (reference), привязанной к имени `r`, и *ячейкой памяти* (cell), на которую указывает эта ссылка.



Если мы сделаем копию `r`, например, связав ее значение с другой переменной, `s`:

```
s = r;
▷ s : Ref Nat
```

то скопируется только ссылка (*стрелка* на диаграмме), а не ячейка:



Это можно проверить, присвоив `s` новое значение:

```
s := 82;
▷ unit : Unit
```

и считав его через `r`:

```
!r;
▷ 82 : Nat
```

Ссылки `r` и `s` называются *псевдонимами* (aliases) одной и той же ячейки.

Упражнение 13.1.1 [★]: Нарисуйте аналогичную диаграмму, показывающую результат выполнения выражений `a = {ref 0, ref 0}` и `b = (λx:Ref Nat. {x,x}) (ref 0)`.

Разделяемое состояние

Наличие псевдонимов сильно усложняет рассуждения о программах, использующих ссылки. Например, выражение `(r:=1;r:=!s)`, которое присваивает значение 1 переменной `r`, а затем немедленно затирает это значение текущим значением переменной `s`, работает точно так же, как простое присваивание `r:=!s`, *за исключением* случая, когда `r` и `s` являются ссылками на одну и ту же ячейку.

Разумеется, вместе с этим, псевдонимы — одно из основных полезных свойств ссылок. В частности, они позволяют организовывать «неявные каналы связи» — разделяемое состояние — между различными частями программы. Например, предположим, что у нас есть ссылочная ячейка и две функции, работающие с ее состоянием:

```
c = ref 0;
▷ c : Ref Nat
incc = λx:Unit. (c := succ (!c); !c);
▷ incc : Unit → Nat
decc = λx:Unit. (c := pred (!c); !c);
▷ decc : Unit → Nat
```

Вызов `incc`

```
incc unit;
▷ 1 : Nat
```

приводит к изменению значения `s`, и это изменение можно наблюдать при вызове `decc`:

```
decc unit;
> 0 : Nat
```

Если мы упакуем `incc` и `decc` в виде записи

```
o = {i = incc, d = decc};
> o : {i:Unit → Nat, d:Unit → Nat}
```

то всю эту структуру можно передавать из функции в функцию как единое целое и с помощью ее компонент увеличивать и уменьшать разделяемую ячейку памяти `s`. В сущности, мы построили простой вариант *объекта* (*object*). Подробно эта идея будет изучена в главе 18.

Ссылки на составные типы

Ссылочная ячейка не обязана содержать всего лишь число: наши примитивы позволяют создавать ссылки на значения произвольного типа, включая функции. Можно, например, при помощи ссылок реализовать (правда, не очень эффективно) массивы чисел. Обозначим именем `NatArray` тип `Ref(Nat → Nat)`.

```
NatArray = Ref (Nat → Nat);
```

Чтобы создать новый массив, выделим ссылочную ячейку и запишем в нее функцию, которая в ответ на любой индекс возвращает 0.

```
newarray = λ_:Unit. ref (λn:Nat. 0);
> newarray : Unit → NatArray
```

Чтобы получить элемент массива, просто применим функцию к нужному индексу.

```
lookup = λa:NatArray. λn:Nat. (!a) n;
> lookup : NatArray → Nat → Nat
```

Интерес в этом кодировании представляет функция `update`. Она принимает массив, индекс и новое значение, которое нужно сохранить в ячейке с этим индексом. Задачу свою она выполняет, создавая (и сохраняя в ссылочной ячейке) новую функцию, которая, будучи спрошена о значении по данному индексу, вернет новое значение, данное в качестве аргумента `update`, а для всех остальных значений индекса передаст задачу поиска функции, которая до сих пор хранилась в ссылочной ячейке.

```
update = λa:NatArray. λm:Nat. λv:Nat.
    let oldf = !a in
        a := (λn:Nat. if equal m n then v else oldf n)
> update : NatArray → Nat → Nat → Unit
```

Упражнение 13.1.2 [★★]: Если бы мы определили функцию `update` более компактно:

```
update = λa:NatArray. λm:Nat. λv:Nat.
          a := (λn:Nat. if equal m n then v else (!a) n;)
```

было бы ее поведение тем же самым?

Ссылки на значения, содержащие другие ссылки, также могут быть очень полезны, и с их помощью можно определять такие структуры данных, как изменяемые списки и деревья. (Как правило, такие структуры также используют *рекурсивные типы* (recursive types), о которых мы поговорим в главе 20.)

Сборка мусора

Последний вопрос, который следует упомянуть, прежде чем мы перейдем к формальному определению ссылок — *освобождение* (deallocation) памяти. Мы не вводим в язык никаких элементарных операций для освобождения ненужных ссылочных ячеек. Вместо этого, как во многих современных языках (включая ML и Java), мы полагаемся на то, что среда исполнения программы проводит *сборку мусора* (garbage collection), собирая и освобождая ячейки, которые перестали быть доступными из программы. Это *не просто* вопрос вкуса в проектировании языков: если в языке имеется явная операция освобождения памяти, то достижение типовой безопасности становится крайне сложной задачей. Причина этому кроется в широко известной проблеме *висячих ссылок* (dangling references): мы выделяем память под ячейку, содержащую число, сохраняем ссылку на нее в некоторой структуре данных, какое-то время ей пользуемся, затем освобождаем ее и выделяем новую ячейку, содержащую булевское значение. При этом, возможно, используется то же самое место в памяти. Теперь у нас может оказаться два имени для одной и той же ячейки памяти — одна с типом `Ref Nat`, а другая с типом `Ref Bool`.

Упражнение 13.1.3 Покажите, как это может привести к нарушению типовой безопасности.

13.2. Типизация

Правила типизации для `ref`, `:=` и `!` прямо следуют из поведения, которого мы от них ожидаем:

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{ref } t_1 : \text{Ref } T_1} \quad (\text{T-REF})$$

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1}{\Gamma \vdash !t_1 : T_1} \quad (\text{T-DEREF})$$

$$\frac{\Gamma \vdash t_1 : \text{Ref } T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \text{Unit}} \quad (\text{T-ASSIGN})$$

13.3. Вычисление

Более тонкие вопросы семантики ссылок возникают, когда мы начинаем формализовывать их операционное поведение. Например, зададим себе вопрос: «Как должны выглядеть *значения* типа `Ref T?`» Основной момент, который нам придется учесть, состоит в том, что вычисление оператора `ref` должно что-то *делать* (а именно, выделять память) и результатом операции должна быть ссылка на эту память.

Так что же такое ссылка?

Память времени исполнения в большинстве реализаций языков программирования представляет собой, в сущности, просто большой массив байтов. Среда времени исполнения следит за тем, какие области в этом массиве используются в каждый момент времени. Когда требуется создать новую ссылочную ячейку, мы выделяем из свободной области памяти сегмент нужного размера (4 байта для ячеек с целыми значениями, 8 байтов для ячеек со значениями типа `Float`, и т. д.), отмечаем этот сегмент как используемый, и возвращаем индекс его начала (как правило, в виде 32- или 64-битного целого). Эти индексы и служат ссылками.

Для наших целей такая конкретика пока не нужна. Мы можем рассматривать память как массив *значений*, а не байтов, абстрагируясь от того, что при исполнении различные значения имеют разные размеры в байтах. Кроме того, можно абстрагироваться от того, что ссылки (т. е., индексы массива) являются числами. Таким образом, мы считаем, что ссылки являются элементами некоторого множества \mathcal{L} *адресов памяти* (store locations), точная природа которого неважна. Всё состояние памяти мы рассматриваем просто как частичную функцию из адресов l в значения. Для состояний памяти мы используем метапеременную μ . Таким образом, ссылка является адресом — абстрактным указателем на память. С этого момента мы используем термин *адрес* (location), а не *ссылка* (reference) или *указатель* (pointer), считая его более абстрактным.⁴

Теперь нужно расширить нашу операционную семантику и включить в нее изменения состояния памяти. Поскольку результат вычисления выражения в общем случае будет зависеть от содержимого памяти в момент вычисления, правила вычисления должны в качестве аргумента принимать не только терм, но и состояние памяти. Более того, поскольку вычисление терма может вызвать побочные эффекты в памяти, эти эффекты могут затем повлиять на вычисление других термов, правила вычисления должны возвращать новое состояние памяти. Таким образом, общий вид одношагового вычисления из $t \rightarrow t'$ превращается в $t \mid \mu \rightarrow t' \mid \mu'$, где μ и μ' — начальное и конечное состояние памяти. В сущности, мы усложнили свое понятие *абстрактной ма-*

⁴ Такой абстрактный подход к адресам не позволяет нам смоделировать *арифметику указателей* (pointer arithmetic) в низкоуровневых языках вроде С. Это намеренное ограничение. Несмотря на то, что арифметика указателей иногда бывает очень полезна (особенно при реализации низкоуровневых компонентов среды времени исполнения, скажем, сборщиков мусора), большинство систем типов не могут с ней формально работать: даже если мы знаем, что адрес n в памяти содержит `Float`, это ничего не говорит о типе значения по адресу $n + 4$. В языке С арифметика указателей — печально известный источник нарушений безопасности типов.

шины (abstract machine) так, что состояние машины состоит теперь не только из указателя команд (представленного в виде терма), но еще и из текущего содержимого памяти.

Чтобы осуществить это изменение, прежде всего нужно дополнить состояниями памяти все имеющиеся правила вычисления:

$$(\lambda x:T_{11}.t_{12})\ v_2 \mid \mu \rightarrow [x \mapsto v_2]t_{12} \mid \mu \quad (\text{E-APPABS})$$

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{t_1\ t_2 \mid \mu \rightarrow t'_1\ t_2 \mid \mu'} \quad (\text{E-APP1})$$

$$\frac{t_2 \mid \mu \rightarrow t'_2 \mid \mu'}{v_1\ t_2 \mid \mu \rightarrow v_1\ t'_2 \mid \mu'} \quad (\text{E-APP2})$$

Обратите внимание, что первое из этих правил возвращает состояние памяти μ неизменным: применение функции само по себе никаких побочных эффектов не имеет. Остальные два правила просто переносят побочные эффекты из предпосылок в заключение.

Затем требуется сделать небольшое дополнение синтаксиса термов. Результатом вычисления выражения `ref` будет новый адрес, поэтому нужно включить адреса в множество возможных результатов вычисления, т. е. в множество значений:

v	$::=$	значения:
$\lambda x:T.t$		значение-абстракция
<code>unit</code>		значение единичного типа
l		адрес в памяти

Поскольку все значения также являются термами, множество термов тоже должно включать адреса.

t	$::=$	термы
x		переменная
$\lambda x:T.t$		абстракция
$t\ t$		применение
<code>unit</code>		константа <code>unit</code>
<code>ref</code> t		порождение ссылки
$!t$		разыменование
$t := t$		присваивание
l		адрес в памяти

Разумеется, такое расширение синтаксиса термов не означает, что мы хотим, чтобы *программисты* писали термы, в которых используются конкретные адреса: такие термы возникают только в качестве промежуточных результатов вычисления. В сущности, слово «язык» в этой главе нужно рассматривать как формализацию *промежуточного языка* (intermediate language), некоторые возможности которого напрямую программистам недоступны.

Пользуясь расширенным синтаксисом, мы можем сформулировать правила вычисления для новых конструкций, которые работают с адресами и содержимым памяти. Во-первых, чтобы вычислить выражение разыменования

$!t_1$, нужно сначала редуцировать терм t_1 , пока он не превратится в значение:

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{!t_1 \mid \mu \rightarrow !t'_1 \mid \mu'} \quad (\text{E-DEREF})$$

Когда t_1 будет сведен к значению, мы получим выражение вида $!l$, где l — некоторый адрес. Терм, пытающийся разыменовать любое другое выражение, скажем, функцию или `unit`, ошибочен. Правила вычисления в этом случае просто объявляют терм тупиковым. Теоремы о типовой безопасности из §13.5 гарантируют нам, что правильно типизированные термы никогда не окажутся в такой ситуации.

$$\frac{\mu(l) = v}{!l \mid \mu \rightarrow v \mid \mu} \quad (\text{E-DEREFLOC})$$

Чтобы вычислить выражение присваивания $t_1 := t_2$, мы сначала вычисляем t_1 , пока он не станет значением (т. е., адресом),

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{t_1 := t_2 \mid \mu \rightarrow t'_1 := t_2 \mid \mu'} \quad (\text{E-ASSIGN1})$$

а затем вычисляем t_2 , пока оно тоже не станет значением (любого вида):

$$\frac{t_2 \mid \mu \rightarrow t'_2 \mid \mu'}{v_1 := t_2 \mid \mu \rightarrow v_1 := t'_2 \mid \mu'} \quad (\text{E-ASSIGN2})$$

Когда мы закончим вычислять t_1 и t_2 , то получим выражение вида $l := v_2$, которое мы выполняем, изменяя содержимое памяти так, чтобы по адресу l содержалось значение v_2 :

$$l := v_2 \mid \mu \rightarrow \text{unit} \mid [l \mapsto v_2] \mu \quad (\text{E-ASSIGN})$$

(Запись $[l \mapsto v_2] \mu$ здесь означает «состояние памяти, где в l содержится v_2 , а по остальным адресам — то же, что и в μ ». Обратите внимание, что на этом шаге вычисления результирующим термом является просто `unit`; интересная часть результата — обновленное состояние памяти.)

Наконец, чтобы вычислить выражение вида `ref` t_1 , мы сначала вычисляем терм t_1 , пока он не превратится в значение:

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{ref } t_1 \mid \mu \rightarrow \text{ref } t'_1 \mid \mu'} \quad (\text{E-REF})$$

Затем, чтобы вычислить сам шаг `ref`, мы выбираем новый адрес l (т. е., адрес, который не входит в текущую область определения μ), и порождаем новое состояние памяти, расширяющее μ новым связыванием $l \mapsto v_1$.

$$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1 \mid \mu \rightarrow l \mid [l \mapsto v_1] \mu} \quad (\text{E-REFV})$$

Терм, получающийся в результате такого шага, — это имя l свежевыделенного адреса в памяти.

Заметим, что наши правила вычисления не производят никакой сборки мусора: мы просто позволяем памяти бесконечно расти по мере вычисления. Это не влияет на правильность результатов вычисления (в конце концов, само определение «мусора» подразумевает, что эта часть содержимого памяти уже недостижима и не играет никакой роли в дальнейшем вычислении). Однако, это означает, что наивная реализация нашего вычислителя иногда будет исчертывать имеющуюся память, в то время как более разумно устроенный вычислитель смог бы продолжить работу, заново используя адреса, содержимое которых превратилось в мусор.

Упражнение 13.3.1 [★★★]: Как можно уточнить наши правила вычисления, чтобы смоделировать сборку мусора? Какую теорему потребуется доказать, чтобы продемонстрировать, что такое уточнение работает правильно?

13.4. Типизация содержимого памяти

После того, как мы расширили синтаксис и правила вычисления, добавив к ним поддержку ссылок, нашей следующей задачей будет написание правил типизации для новых конструкций — и, разумеется, проверка их непротиворечивости. Естественно, первый вопрос будет: «Каков тип адреса?».

При вычислении терма, содержащего конкретные адреса, тип результата зависит от содержимого памяти в начале вычисления. Например, если мы вычисляем терм $!l_2$ при состоянии памяти $(l_1 \mapsto \text{unit}, l_2 \mapsto \text{unit})$, то результатом будет `unit`; если тот же самый терм вычисляется при состоянии памяти $(l_1 \rightarrow \text{Unit}, l_2 \rightarrow \lambda x:\text{Unit}.x)$, то результатом будет $\lambda x:\text{Unit}.x$. По отношению к первому из этих состояний содержимое адреса l_2 имеет тип `Unit`, по отношению ко второму — тип `Unit → Unit`. Это наблюдение немедленно приводит к первой попытке построения правила типизации для адресов:

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \text{Ref } T_1}$$

То есть, чтобы найти тип адреса l , мы смотрим на текущее содержимое ячееки с этим адресом и вычисляем тип T_1 этого содержимого. Типом адреса в этом случае будет `Ref` T_1 .

Начав таким образом, нам придется еще немного поработать, чтобы добиться непротиворечивого описания. В сущности, сделав тип терма зависимым от состояния памяти, мы превратили трехместное отношение типизации (между контекстами, термами и типами) в четырехместное (между контекстами, *содержимым памяти*, термами и типами). Интуитивно ясно, что состояние памяти является частью контекста, в котором мы вычисляем тип терма. Поэтому мы будем записывать это состояние слева от символа \vdash : $\Gamma \mid \mu \vdash t : T$. Правило типизации для ссылок теперь имеет такой вид:

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T_1}{\Gamma \mid \mu \vdash l : \text{Ref } T_1}$$

и все прочие правила типизации в системе также дополняются, чтобы учесть состояние памяти. Остальные правила не делают с этим состоянием ничего интересного — они просто передают его от предпосылки к заключению.

Однако с этим правилом возникают две проблемы. Во-первых, проверка типов становится довольно неэффективной, поскольку вычисление типа адреса l включает вычисление типа текущего состояния v по адресу l . Если l встречается в терме t много раз, мы будем многократно пересчитывать тип v в процессе вывода дерева типизации для t . Хуже того, если сам v содержит адреса, нам придется и их тип пересчитывать каждый раз, когда мы их встретим. Например, если состояние памяти таково:

$$\begin{aligned} l_1 &\mapsto \lambda x:\text{Nat}. \ 999, \\ l_2 &\mapsto \lambda x:\text{Nat}. \ (!l_1) \ x, \\ l_3 &\mapsto \lambda x:\text{Nat}. \ (!l_2) \ x, \\ l_4 &\mapsto \lambda x:\text{Nat}. \ (!l_3) \ x, \\ l_5 &\mapsto \lambda x:\text{Nat}. \ (!l_4) \ x, \end{aligned}$$

то вычисление типа l_5 требует вычисления типов l_4 , l_3 , l_2 и l_1 .

Во-вторых, предложенное правило типизации может вообще оказаться неспособным что-либо вывести, если в памяти имеется *цикл* (cycle). Например, нет ни одного конечного дерева вывода типизации для адреса l_2 , если содержимое памяти таково:

$$\begin{aligned} l_1 &\mapsto \lambda x:\text{Nat}. \ (!l_2) \ x \\ l_2 &\mapsto \lambda x:\text{Nat}. \ (!l_1) \ x, \end{aligned}$$

поскольку вычисление типа для l_2 требует нахождения типа для l_1 , который, в свою очередь, обращается к l_2 , и т. д. Такие циклические структуры ссылок действительно иногда встречаются на практике (скажем, с их помощью можно построить двусвязные списки), а нам хотелось бы, чтобы наша система типов умела работать с такими данными.

Упражнение 13.4.1 [★]: *Можете ли вы найти терм, который при вычислении создал бы такое циклическое состояние памяти?*

Обе эти проблемы возникают из-за того, что предложенное нами правило типизации для адресов требует пересчитывать тип адреса каждый раз, когда он упоминается в терме. Однако интуитивно ясно, что это необязательно. В конце концов, когда адрес впервые создается, мы знаем, каков тип начального значения, которое мы записываем по этому адресу. Более того, несмотря на то, что впоследствии мы можем сохранять в той же ячейке другие значения, эти другие значения всегда будут того же типа, что и начальное. Другими словами, для каждого адреса в памяти мы всегда имеем в виду один конкретный тип, определяемый в момент выделения памяти. Эти типы можно собрать вместе в *структуру типизации памяти* (store typing) — конечную функцию, отображающую адреса в типы. Для обозначения таких функций мы будем использовать метапеременную Σ .

Предположим, что нам *дана* структура типизации памяти Σ , описывающая состояние памяти μ , по отношению к которому будет вычислен терм t .

Тогда с помощью Σ мы можем вычислить тип результата t , не обращаясь прямо к μ . Например, если Σ такова: ($l_1 \mapsto \text{Unit}, l_2 \mapsto \text{Unit} \rightarrow \text{Unit}$), то мы можем немедленно заключить, что $!l_2$ имеет тип $\text{Unit} \rightarrow \text{Unit}$. В общем случае, можно переформулировать правило типизации для адресов через структуры типизации памяти таким образом:

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \text{Ref } T_1} \quad (\text{T-Loc})$$

Типизация опять оказывается четырехместным отношением, но параметром служит *структура типизации* памяти, а не конкретное ее содержимое. Остальные правила типизации дополняются аналогичным образом с учетом структур типизации памяти.

Разумеется, такие правила типизации будут правильно предсказывать результаты вычисления только в том случае, если конкретное состояние памяти во время вычисления действительно будет соответствовать типизации памяти, которую мы предполагаем в рамках задачи проверки типов. Это условие в точности повторяет ситуацию со свободными переменными во всех исчислениях, рассмотренных нами до сих пор: лемма о подстановке (9.3.8) говорит нам, что, если $\Gamma \vdash t : T$, то мы можем заменить свободные переменные t значениями типов, указанных в Γ , и получим замкнутый терм типа T , который, согласно теореме о сохранении типов (9.3.9), при вычислении даст окончательный результат типа T (если какой-либо результат вообще получится). В §13.5 мы увидим, как формализовать подобную интуицию для состояний памяти и структур типизации памяти.

Заметим, наконец, что при проверке типов для термов, которые пишут программисты, нам не требуется прибегать ни к каким хитростям, чтобы вычислить, какую типизацию памяти нужно использовать. Как указано выше, конкретные адреса-константы возникают только в термах-промежуточных результатах вычислений; они находятся вне языка, используемого программистами при написании программ. Таким образом, в термах программистов мы можем проверять типизацию относительно *пустой* структуры типизации памяти. По мере хода вычисления и создания новых адресов всегда можно увидеть, как расширить типизацию памяти, глядя на типы начальных значений, размещаемых в свежевыделенных ячейках; это интуитивное рассуждение будет формализовано в теореме о сохранении типов (13.5.3).

Теперь, когда мы разобрались с адресами, сформулировать правила типизации для остальных новых синтаксических форм достаточно просто. Когда мы создаем ссылку на значение типа T_1 , сама ссылка имеет тип $\text{Ref } T_1$.

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \text{ref } t_1 : \text{Ref } T_1} \quad (\text{T-REF})$$

Заметим, что здесь нам не нужно расширять структуру типизации памяти, поскольку *имя* новой ячейки не будет определено вплоть до времени исполнения, а в Σ записывается только соотношение между уже выделенными ячейками памяти и их типами.

Соответственно, если t_1 имеет значением адрес типа $\text{Ref } T_{11}$, то разыменование этого терма всегда даст значение типа T_{11} .

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \quad (\text{T-DEREF})$$

Наконец, если t_1 обозначает ячейку типа $\text{Ref } T_{11}$, в этой ячейке можно сохранить t_2 при условии, что t_2 также имеет тип T_{11} :

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}} \quad (\text{T-ASSIGN})$$

На рис. 13.1 изображены правила типизации (а также, для справки, синтаксические правила и правила вычисления) простого типизированного лямбда-исчисления со ссылками.

13.5. Безопасность

Последняя наша задача в этой главе — убедиться в том, что стандартные характеристики типовой безопасности продолжают выполняться для исчисления, включающего в себя ссылки. Теорему о продвижении («правильно типизированные термы не находятся в тупике») можно сформулировать и доказать почти так же, как и ранее (см. 13.5.7); требуется лишь рассмотреть в доказательстве несколько новых несложных вариантов, относящихся к новым конструкциям. Теорема о сохранении немного интереснее, так что мы сначала обратимся к ней.

Поскольку мы расширили как отношение вычисления (начальным и конечным состояниями памяти), так и отношение типизации (структурой типизации памяти), требуется изменить текст теоремы о сохранении, упомянув эти параметры. Ясно, впрочем, что нельзя просто добавить состояния памяти и структуры типизации, не сказав ничего о том, как они связаны между собой.

Если $\Gamma \mid \Sigma \vdash t : T$ и $t \mid \mu \rightarrow t' \mid \mu'$, то $\Gamma \mid \Sigma \vdash t' : T$ (*Неверно!*)

Если мы проверим типы, исходя из некоторых предположений о типах значений в памяти, а затем начнем выполнять программу по отношению к памяти, где эти предположения нарушаются, результат будет катастрофическим. Следующее требование выражает нужное нам ограничение:

Определение 13.5.1 *Состояние памяти μ называют правильно типизированным (well typed) относительно типового контекста Γ и структуры типизации памяти Σ , что записывается в виде $\Gamma \mid \Sigma \vdash \mu$, если $\text{dom}(\mu) = \text{dom}(\Sigma)$ и $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ для каждого $l \in \text{dom}(\mu)$.*

Говоря неформально, состояние памяти μ корректно по отношению к структуре типизации Σ , если каждое значение в памяти имеет тип, предписываемый ему структурой типизации.

Упражнение 13.5.2 [★★]: Можете ли вы найти контекст Γ , состояние памяти μ и две различных структуры типизации Σ_1 и Σ_2 , так, чтобы одновременно выполнялось $\Gamma \mid \Sigma_1 \vdash \mu$ и $\Gamma \mid \Sigma_2 \vdash \mu$?

Теперь мы можем сформулировать утверждение, более близкое к необходимому свойству сохранения:

Если

$$\Gamma \mid \Sigma \vdash t : T$$

$$t \mid \mu \rightarrow t' \mid \mu'$$

$$\Gamma \mid \Sigma \vdash \mu$$

то $\Gamma \mid \Sigma \vdash t' : T$. (Лучше, но все еще неверно.)

Это утверждение работает для всех правил вычисления, кроме правила выделения памяти E-REFV. Проблема здесь в том, что в результате выполнения правила получается состояние памяти с более широкой областью определения, чем в исходном состоянии, и из-за этого оказывается ложным заключение нашего утверждения: если μ' включает связывание с новым адресом l , то l не может быть в области определения Σ , и утверждение, что терм t' (который наверняка упоминает l) типизируем относительно Σ , будет неверным.

Ясно, что поскольку состояние памяти в процессе вычисления может расширяться, нужно разрешить расширение и для структуры типизации Σ . Отсюда следует окончательная (верная) формулировка свойства сохранения:

Теорема 13.5.3 [Сохранение]: Если

$$\Gamma \mid \Sigma \vdash t : T$$

$$\Gamma \mid \Sigma \vdash \mu$$

$$t \mid \mu \rightarrow t' \mid \mu'$$

то для некоторого $\Sigma' \supseteq \Sigma$,

$$\Gamma \mid \Sigma' \vdash t' : T$$

$$\Gamma \mid \Sigma' \vdash \mu.$$

Заметим, что теорема о сохранении утверждает только, что существует некоторое состояние памяти $\Sigma' \supseteq \Sigma$ (т. е., совпадающее с Σ на всех старых адресах), такое, что новый терм t' правильно типизирован относительно Σ' ; она ничего не говорит о том, как именно Σ' выглядит. Неформально, разумеется, понятно, что Σ' либо совпадает с Σ , либо равняется в точности $(\Sigma, l \mapsto T_1)$, где l — вновь выделенный адрес (новый элемент области определения μ'), а T_1 — тип начального значения, хранимого по адресу l в расширенном состоянии памяти $(\mu, l \mapsto v_1)$, однако явное внесение этих подробностей усложнило бы утверждение теоремы, не принося никакой дополнительной пользы: более слабая версия, приведенная выше, уже сформулирована верно (поскольку ее заключение следует из предпосылок) и достаточна для того, чтобы через ее многократное применение заключить, что всякая последовательность шагов вычисления сохраняет правильную типизацию. В сочетании со свойством продвижения мы получаем обычную гарантию того, что «правильно типизированные программы никогда не ломаются».

Для доказательства сохранения нам понадобится еще несколько промежуточных лемм. Первая из них — несложное расширение стандартной леммы о подстановке (9.3.8).

Лемма 13.5.4 [Подстановка]: Если $\Gamma, x:S \mid \Sigma \vdash t : T$ и $\Gamma \mid \Sigma \vdash s : S$, то $\Gamma \mid \Sigma \vdash [x \mapsto s]t : T$.

Доказательство: В точности как в лемме 9.3.8.

Следующая лемма утверждает, что замена содержимого ячейки памяти новым значением соответствующего типа не изменяет общую структуру типизации памяти.

Лемма 13.5.5 Если

$$\begin{aligned} \Gamma \mid \Sigma &\vdash \mu \\ \Sigma(l) &= T \\ \Gamma \mid \Sigma &\vdash v : T \\ \text{то } \Gamma \mid \Sigma &\vdash [l \mapsto v]\mu. \end{aligned}$$

Доказательство: Непосредственно из определения отношения $\Gamma \mid \Sigma \vdash \mu$.

Наконец, нужна своего рода лемма об ослаблении для состояний памяти, утверждающая, что, если память расширена новым адресом, расширенное состояние памяти по-прежнему позволяет приписывать старым термам те же типы, что и исходное.

Лемма 13.5.6 Если $\Gamma \mid \Sigma \vdash t : T$ и $\Sigma' \supseteq \Sigma$, то $\Gamma \mid \Sigma' \vdash t : T$.

Доказательство: Несложная индукция.

Теперь мы можем доказать основную теорему о сохранении:

Доказательство 13.5.3: Несложная индукция по деревьям вывода вычисления с использованием вышеупомянутых лемм и свойства инверсии для правил типизации (это прямолинейное расширение свойства 9.3.1).

Утверждение теоремы о продвижении (9.3.5) также требуется расширить, чтобы учесть состояния памяти и структуры типизации:

Теорема 13.5.7 [ПРОДВИЖЕНИЕ]: Предположим, что t — замкнутый, правильно типизированный терм (то есть, $\emptyset \mid \Sigma \vdash t : T$) для некоторых T и Σ . Тогда либо t является значением, либо для любого состояния памяти μ , такого что $\emptyset \mid \Sigma \vdash \mu$, существуют терм t' и состояние памяти μ' , такие, что $t \mid \mu \rightarrow t' \mid \mu'$.

Доказательство: Прямолинейная индукция по деревьям вывода типов, по той же схеме, что и доказательство теоремы 9.3.5. (Лемма о канонических формах 9.3.4 требует двух дополнительных утверждений: утверждения о том, что все значения типа *Ref* являются адресами, и аналогичного утверждения для *Unit*.)

Упражнение 13.5.8 [РЕКОМЕНДУЕТСЯ, ★★]: Является ли отношение вычисления из этой главы нормализующим для правильно типизированных термов? Если да, докажите это. Если нет, напишите правильно типизированную функцию вычисления факториала в нашем текущем исчислении (расширенном числами и булевскими значениями).

13.6. Дополнительные замечания

Способ подачи материала, выбранный в этой главе, основан на подходе Харпера (Harper, 1994, 1996). Описание в аналогичном стиле дано в статье Райта и Феллайсена (Wright and Felleisen, 1994).

Сочетание ссылок (или других вычислительных эффектов) с полиморфным выводом типов в стиле ML ведет к некоторым довольно тонким проблемам (см. §22.7). Этим вопросам уделялось немалое внимание в теоретической литературе. См. труды Тофте (Tofte, 1990), Хоанга и др. (Hoang, Mitchell, and Viswanathan, 1993), Жувело и Гиффорда (Jouvelot and Gifford, 1991), Талпина и Жувело (Talpin and Jouvelot, 1992), Леруа и Вайса (Leroy and Weis, 1991), Райта (Wright, 1992), Харпера (Harper, 1994, 1996), а также упомянутые в них источники.

Статическое предсказание возможного наличия псевдонимов — давно известная проблема как в построении компиляторов (известная как *анализ псевдонимов*, alias analysis), так и в теории языков программирования. В оказавшей большое влияние ранней работе Рейнольдса по этой проблеме (Reynolds, 1978, 1989) введен термин *синтаксический контроль паразитных взаимодействий* (syntactic control of interference). Его идеи недавно вызвали новый всплеск активности — см. работы О'Херна и др. (O'Hearn, Takeyama, Power, and Tennent, 1995), а также Смита и др. (Smith, Walker, and Morrisett, 2000). Более общие подходы к исследованию псевдонимов описываются в работах Рейнольдса (Reynolds, 1981) и Иштиака и О'Херна (Ishtiaq and O'Hearn, 2001). См. также литературу, процитированную в этих работах.

Подробное обсуждение методов сборки мусора можно найти у Джонса и Линса (Jones and Lins, 1996). Более семантически-ориентированный подход представлен работой Моррисета и др. (Morrisett, Felleisen, and Harper, 1995).

Найти причину этого эффекта
Или, верней, дефекта, потому
что
Дефектный сей эффект
небеспринчен.

Гамлет II, ii, 101 (перевод
М. Лозинского)

Палец, указывающий на луну —
не луна.

Буддистская пословица

→ Unit Ref

Расширяет λ_{\rightarrow} с типом Unit (9.1 и 11.2)

Синтаксис

$t ::=$	термы
x	переменная
$\lambda x:T.t$	абстракция
$t t$	применение
unit	константа unit
$\text{ref } t$	создание ссылки
$!t$	разыменование
$t := t$	присваивание
l	адрес в памяти

$v ::=$	значения:
$\lambda x:T.t$	значение-абстракция
unit	константа unit
l	адрес в памяти

$T ::=$	типы:
$T \rightarrow T$	тип функций
Unit	единичный тип
$\text{Ref } T$	тип ссылочных ячеек

$\Gamma ::=$	контексты:
\emptyset	пустой контекст
$\Gamma, x:T$	связывание термовой переменной

$\mu ::=$	состояния памяти:
\emptyset	пустое
$\mu, l \rightarrow v$	связывание адреса

$\Sigma ::=$	структуры типизации памяти:
\emptyset	пустая
$\Sigma, l:T$	тиปизация адреса

Вычисление

$$t | \mu \rightarrow t' | \mu'$$

$$\frac{t_1 | \mu \rightarrow t'_1 | \mu'}{t_1 \ t_2 | \mu \rightarrow t'_1 \ t_2 | \mu'} \quad (\text{E-APP1})$$

$$\frac{t_2 | \mu \rightarrow t'_2 | \mu'}{v_1 \ t_2 | \mu \rightarrow v_1 \ t'_2 | \mu'} \quad (\text{E-APP2})$$

$$(\lambda x:T_{11}.t_{12}) \ v_2 | \mu \rightarrow [x \mapsto v_2]t_{12} | \mu \quad (\text{E-APPABs})$$

$$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1 | \mu \rightarrow l | (\mu, l \mapsto v_1)} \quad (\text{E-REFV})$$

$$\frac{t_1 | \mu \rightarrow t'_1 | \mu'}{\text{ref } t_1 | \mu \rightarrow \text{ref } t'_1 | \mu'} \quad (\text{E-REF})$$

$$\frac{\mu(l) = v}{!l | \mu \rightarrow v | \mu} \quad (\text{E-DEREFLOC})$$

$$\frac{t_1 | \mu \rightarrow t'_1 | \mu'}{!t_1 | \mu \rightarrow !t'_1 | \mu'} \quad (\text{E-DEREF})$$

$$l := v_2 | \mu \rightarrow \text{unit} | [l \mapsto v_2]\mu \quad (\text{E-ASSIGN})$$

$$\frac{t_1 | \mu \rightarrow t'_1 | \mu'}{t_1 := t_2 | \mu \rightarrow t'_1 := t_2 | \mu'} \quad (\text{E-ASSIGN1})$$

$$\frac{t_2 | \mu \rightarrow t'_2 | \mu'}{v_1 := t_2 | \mu \rightarrow v_1 := t'_2 | \mu'} \quad (\text{E-ASSIGN2})$$

Типизация	$\Gamma \mid \Sigma \vdash t : T$	
	$\frac{x : T \in \Gamma}{\Gamma \mid \Sigma \vdash x : T}$	(T-VAR)
	$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \text{Ref } T_1}$	(T-LOC)
	$\frac{\Gamma, x:T_1 \mid \Sigma \vdash t_2 : T_2}{\Gamma \mid \Sigma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2}$	(T-ABS)
	$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \text{ref } t_1 : \text{Ref } T_1}$	(T-REF)
	$\frac{\Gamma \mid \Sigma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \mid \Sigma \vdash t_2}{\Gamma \mid \Sigma \vdash t_1 \ t_2 : T_{12}}$	(T-APP)
	$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}}$	(T-DEREF)
$\Gamma \mid \Sigma \vdash \text{unit} : \text{Unit}$	$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}}$	(T-ASSIGN)

Рис. 13.2. Ссылки (*продолжение*)

Глава 14

Исключения

В главе 13 мы увидели, как можно расширить операционную семантику простого типизированного лямбда-исчисления изменяемыми ссылками. Также мы исследовали, как изменяются правила типизации и доказывали свойства типовой безопасности при таком расширении. В этой главе мы рассматриваем другое расширение нашей исходной вычислительной модели: порождение и обработку исключений.

В практическом программировании на каждом шагу встречаются ситуации, в которых функция должна сообщить вызывающей программе, что по какой-либо причине она не может выполнить свою задачу: из-за того, что какое-то вычисление сталкивается с делением на ноль или арифметическим переполнением, из-за того, что ключ поиска отсутствует в словаре, или из-за того, что произошло какое-либо аварийное событие — кончилась системная память или пользователь прервал процесс.

В некоторых случаях можно подать сигнал об этих исключительных ситуациях, используя функцию с вариантным возвращаемым значением, как мы видели в §11.10. Но если исключительная ситуация действительно *исключительна*, нехорошо заставлять программиста при каждом вызове нашей функции учитывать возможность ее возникновения. Лучше, если исключительные обстоятельства вызовут немедленную передачу управления *обработчику исключений*, определенному в программе на более высоком уровне, или же, если исключительная ситуация достаточно редка или если вызывающая подпрограмма все равно никак не сможет с ней справиться, просто вызовут прекращение выполнения программы. Мы рассмотрим сначала этот последний вариант (§14.1), в котором всякое исключение немедленно останавливает программу, затем добавим механизм для перехвата и обработки исключений (§14.2), и, наконец, усовершенствуем оба эти механизма, обеспечив передачу дополнительной информации, определяемой программистом, между точкой возникновения исключения и обработчиком (§14.3).

Системы, изучаемые в этой главе — простое типизированное лямбда-исчисление (рис. 9.1), расширенное различными примитивами для порождения и обработки исключений (рис. 14.1 и 14.2). Реализация первого из этих расширений на OCaml называется **fullerror**. Язык, в котором исключения сопровождаются значениями (рис. 14.3), не реализован.

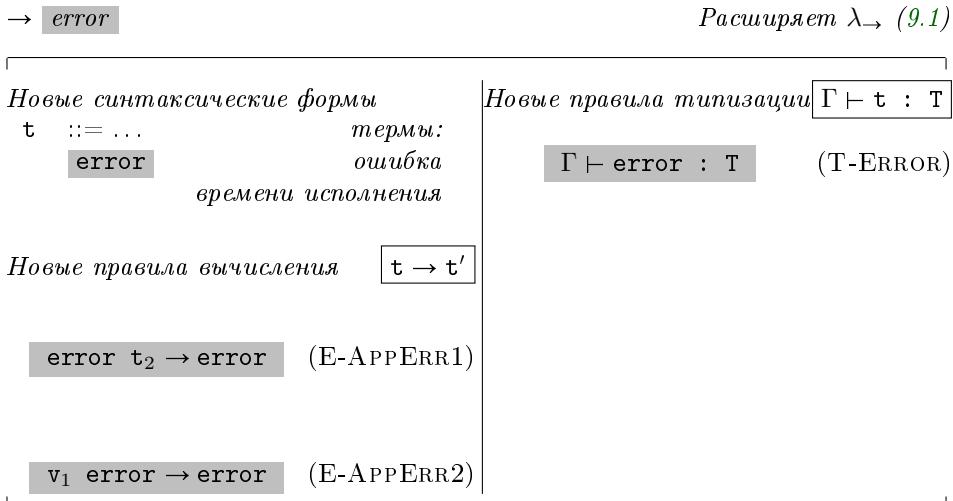


Рис. 14.1. Ошибки

14.1. Порождение исключений

Для начала добавим в лямбда-исчисление простейший из возможных механизмов оповещения об исключениях: терм **error** («ошибка»), который, будучи выполнен, прекращает вычисление терма, в котором он содержится. Необходимые дополнения изображены на рис. 14.1.

Основное проектное решение при разработке правил для терма **error** — это вопрос о том, как следует формализовать «ненормальное завершение» в нашей операционной семантике. Мы используем простейший вариант: результатом прерванной программы служит сам терм **error**. Правила E-APPER1 и E-APPER2 реализуют это поведение. Правило E-APPER1 говорит, что если мы встречаем терм **error** при попытке вычисления левой части терма-применения, мы должны немедленно возвратить **error** в качестве результата применения. Точно так же, E-APPER2 говорит, что если мы встречаем терм **error**, попытавшись вычислить значение аргумента применения, мы должны прекратить обрабатывать это применение и немедленно вернуть **error**.

Обратите внимание, что мы не включили **error** в список значений — только в список термов. Это исключает конфликт между левыми частями правил E-APPABS и E-APPER2 — т. е., не будет неоднозначности по поводу того, следует ли вычислить терм

$(\lambda x:\text{Nat}. 0) \text{ error}$

как применение (и вернуть в результате 0) или прервать выполнение программы: возможен только второй из этих вариантов. Заметим также, что в E-APPER2 мы использовали метапеременную v_1 (а не t_1 , имеющую значением произвольные термы). Это заставляет вычислитель ждать, пока левая часть терма-применения будет вычислена, прежде чем прервать исполнение, даже если в правой части стоит **error**. Таким образом, терм вида

```
(fix (λx:Mat.x)) error
```

приведет не к ошибке, а к зацикливанию. Благодаря этим условиям наше отношение вычисления остается детерминистским.

Правило типизации T-ERROR также представляет интерес. Поскольку порождение исключений может потребоваться в любом контексте, терму `error` разрешается иметь какой угодно тип. В терме

```
(λx:Boolean.x) error
```

он имеет тип `Boolean`. В терме

```
(λx:Boolean.x) (error true)
```

он имеет тип `Boolean → Boolean`.

Такая гибкость в типизации `error` приводит к некоторым трудностям при реализации алгоритма проверки типов, поскольку при этом теряется свойство, гласящее, что у каждого терма только один тип (теорема 9.3.3). С этим можно справиться различными методами. В языке с подтипами мы можем дать терму `error` наименьший тип `Bot` (см. §15.4), который при необходимости *повышается* (*promotes*) до любого типа. В языке с параметрическим полиморфизмом (см. главу 23) терму `error` можно дать полиморфный тип $\forall X.X$, который *конкретизируется* (*instantiates*) любым другим типом. В обоих случаях бесконечное число возможных типов для `error` компактно представлено при помощи одного типа.

Упражнение 14.1.1 [★]: Не будет ли проще потребовать от программиста при каждом использовании помечать `error` его предполагаемым типом?

Свойство сохранения для языка с исключениями выглядит так же, как всегда: если терм имеет тип T , и мы позволяем ему пройти один шаг вычисления, результат по-прежнему будет иметь тип T . Однако это свойство продвижения требуется несколько уточнить. В исходной форме оно говорило, что правильно типизированная программа обязана при вычислении выдать значение (или зациклиться). Однако теперь мы ввели нормальную форму `error`, которая не является значением, но безусловно может служить результатом исполнения правильно типизированной программы. Свойство продвижения нужно переформулировать с учетом этого обстоятельства.

Теорема 14.1.2 [ПРОДВИЖЕНИЕ]: Пусть имеется замкнутый, правильно типизированный терм t в нормальной форме. Тогда либо t — значение, либо $t = \text{error}$.

14.2. Обработка исключений

Правила вычисления для `error` можно рассматривать как «откатывание стека вызовов» с отбрасыванием ждущих вызовов функций до тех пор, пока ошибка не окажется на самом верхнем уровне. В настоящих реализациях языков с исключениями именно это и происходит: стек вызовов состоит из множества записей активации (activation records), по одной на каждый активный

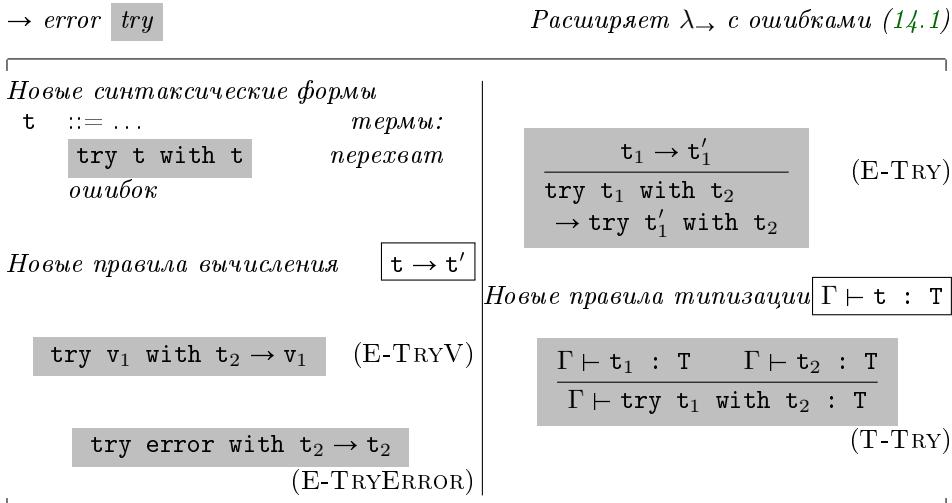


Рис. 14.2. Обработка ошибок

вызов функции; когда возникает исключение, записи активации снимаются со стека вызовов, пока он не опустеет.

В большинстве языков с исключениями также можно установить в стеке вызовов *обработчики исключений* (exception handlers). Когда возникает исключение, записи активации снимаются со стека вызовов до тех пор, пока на вершине не окажется обработчик исключений, и тогда вычисление продолжается с этого обработчика. Другими словами, исключение работает как нелокальная передача управления, целью которой является установленный последним (т. е. ближайший на стеке вызовов) обработчик исключений.

Наша формулировка обработчиков исключений, приведенная на рис. 14.2, похожа как на ML, так и на Java. Выражение $\text{try } t_1 \text{ with } t_2$ означает «вернуть результат вычисления t_1 , если при этом не возникнет ошибки; иначе выполнить обработчик t_2 ». Правило вычисления E-TRYV говорит, что если терм t_1 свелся к значению v_1 , то конструкцию try можно отбросить, поскольку мы уже знаем, что она не понадобится. С другой стороны, правило E-TRYERROR говорит, что если при вычислении t_1 получилась ошибка error , следует заменить конструкцию try термом t_2 и продолжить его вычисление. E-TRY говорит, что до тех пор, пока t_1 не свелся ни к значению, ни к error , следует просто продолжать его вычислять, не трогая t_2 .

Правило типизации для try прямо следует из его операционной семантики. Результатом всей конструкции try может быть либо результат основной части t_1 , либо результат обработчика t_2 ; нужно просто потребовать, чтобы оба они имели один и тот же тип T , который также будет типом всего try .

Свойство типовой безопасности и его доказательство, по существу, остаются неизменными по сравнению с предыдущим разделом.

→ исключения

Расширяем λ_{\rightarrow} (9.1)

Новые синтаксические формы

$t ::= \dots$	термы:
raise t	порождение исключений
try t with t	обработка исключений

Новые правила вычисления

$$t \rightarrow t'$$

$$(raise v_{11}) \ t_2 \rightarrow raise v_{11} \quad (\text{E-APPRAISE1})$$

$$v_1 \ (raise v_{21}) \rightarrow raise v_{21} \quad (\text{E-APPRAISE2})$$

$$\frac{t_1 \rightarrow t'_1}{raise t_1 \rightarrow raise t'_1} \quad (\text{E-RAISE})$$

$$raise (raise v_{11}) \rightarrow raise v_{11} \quad (\text{E-RAISERaise})$$

$$try v_1 \ with \ t_2 \rightarrow v_1 \quad (\text{E-TRYV})$$

$$try raise v_{11} \ with \ t_2 \rightarrow t_2 \ v_{11} \quad (\text{E-TRYRAISE})$$

$$\frac{t_1 \rightarrow t'_1}{try t_1 \ with \ t_2 \rightarrow try t'_1 \ with \ t_2} \quad (\text{E-TRY})$$

Новые правила типизации

$$\frac{\Gamma \vdash t_1 : T_{exn}}{\Gamma \vdash raise t_1 : T} \quad (\text{T-EXN})$$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_{exn} \rightarrow T}{\Gamma \vdash try t_1 \ with \ t_2 : T} \quad (\text{T-TRY})$$

Рис. 14.3. Исключения, сопровождаемые значениями

14.3. Исключения, сопровождаемые значениями

Механизмы, введенные нами в §14.1 и §14.2, позволяют функциям сообщить вызывающей программе, что «случилось нечто необычное». Как правило, полезно бывает также отсылать некоторую дополнительную информацию о том, *какое именно* необычное событие произошло, поскольку действие, которое должен предпринять обработчик — попытка исправить ситуацию и вызвать функцию заново, либо выдача пользователю внятного сообщения об ошибке, — может зависеть от этой информации.

На рис. 14.3 показано, как наши базовые конструкции для обработки ошибок могут быть расширены, чтобы каждое исключение было снабжено значением. Тип этого значения записывается в виде T_{exn} . Пока что мы оставляем открытым вопрос о том, что это за тип; ниже обсуждается несколько вариантов.

Атомарный терм `error` заменяется конструктором терма `raise t`, где t —

дополнительная информация, которую мы хотим передать обработчику исключений. Синтаксис конструкции `try` остается прежним, но обработчик t_2 в `try t1 with t2` теперь интерпретируется как *функция*, принимающая в качестве аргумента дополнительную информацию.

Правило вычисления E-TRYRAISE реализует нужное нам поведение, принимая дополнительную информацию, сопровождающую `raise`, из тела t_1 , и передавая ее обработчику t_2 . Правила E-APPROISE1 и E-APPROISE2 проносят исключения через применения, в точности как E-APPERR1 и E-APPERR2 на рис. 14.1. Заметим, однако, что эти дополнительные правила могут распространять только исключения, в которых дополнительная информация является значением; если мы попробуем вычислить `raise` с дополнительной информацией, которая сама требует вычисления, эти правила окажутся заблокированы, и нам придется вычислять дополнительную информацию при помощи E-RAISE. Правило E-RAISERAISE распространяет исключения, которые могут возникнуть *во время* вычисления дополнительной информации, предназначеннной для отсылки с каким-то другим исключением. E-TRYV говорит, что мы можем отбросить `try`, если его основное тело уже свелось к значению, так же, как мы это делали в §14.2. E-TRY указывает, что вычислитель должен работать с телом `try`, пока оно не вернуло либо значение, либо `raise`.

Правила типизации отражают эти изменения в поведении конструкций. В T-RAISE мы требуем, чтобы дополнительная информация имела тип T_{exn} ; все выражение `raise` целиком получает любой тип T , какого требует контекст. В правиле T-TRY мы проверяем, что обработчик t_2 является функцией, которая, получая дополнительную информацию типа T_{exn} , выдает результат того же типа, что и t_1 .

Теперь, наконец, рассмотрим возможные альтернативы для типа T_{exn} .

1. Можно принять в качестве T_{exn} просто тип `Nat`. Это соответствует соглашению `errno`, используемому, например, в функциях операционной системы Unix: каждый системный вызов возвращает числовой «код ошибки», причем 0 обозначает успех, а другие значения сообщают о различных ошибочных ситуациях.
2. Можно принять в качестве T_{exn} тип `String`. Это избавляет от необходимости поиска номеров ошибок в таблицах, а код, порождающий исключение, может при желании генерировать достаточно подробное описание ситуации. Ценой такой дополнительной гибкости будет то, что обработчикам ошибок, возможно, придется *разбирать* эти строки, чтобы понять, что произошло.
3. Можно сохранить возможность передавать информативные исключения, избегая при этом разбора строк, если объявить T_{exn} как *многовариант* (variant type):

```
 $T_{exn} = <\text{divideByZero: Unit,}$ 
        $\text{overflow: Unit,}$ 
        $\text{fileNotFound: String,}$ 
        $\text{fileNotReadable: String,}$ 
       ...>
```

Такая схема позволяет обработчику различать виды исключений при помощи выражения `case`. Кроме того, различные виды исключений могут нести различные типы дополнительной информации: в исключениях вроде `divideByZero` ничего лишнего не требуется, вместе с `FileNotFoundException` можно передать имя файла, который пытались открыть, когда возникла ошибка, и т. д.

Проблема с этим вариантом состоит в том, что он недостаточно гибок и требует, чтобы мы *заранее* задали множество исключений, которые может породить любая программа (т. е., множество тегов вариантового типа T_{exn}). Программистам не дается возможности определения собственных исключений.

4. Ту же идею можно улучшить, обеспечив возможность создания пользовательских типов исключений, если объявить T_{exn} *расширяемым вариантым типом* (extensible variant type). Так сделано в ML, в котором имеется единственный расширяемый вариантыкий тип `exp`.¹ Объявление `exception l of T` в языке ML можно в нашей терминологии понимать так: «убедиться, что l отличается от всех тегов, уже имеющихся в вариантыком типе T_{exn} ² и, начиная с этого момента, считать, что T_{exn} равен $\langle l_1:T_1, \dots l_n:T_n, l:T \rangle$, где варианты от $l_1:T_1$ до $l_n:T_n$ были возможны до данного объявления».

Синтаксис ML для порождения исключений выглядит как `raise l(t)`, где l — тег исключения, определенный в текущей области видимости. Эту конструкцию можно понимать как сочетание оператора навешивания тега и нашего обычного `raise`:

$$\text{raise } l(t) \stackrel{\text{def}}{=} \text{raise } (\langle l=t \rangle \text{ as } T_{exn})$$

Аналогичным образом, конструкция `try` языка ML может быть записана без сахара в виде простого `try` в сочетании с `case`.

$$\begin{aligned} \text{try } t \text{ with } l(x) \rightarrow h &\stackrel{\text{def}}{=} \text{try } t \text{ with} \\ &\quad \lambda e:T_{exn}. \text{ case } e \text{ of} \\ &\quad \quad \langle l=x \rangle \Rightarrow h \\ &\quad \quad | _ \Rightarrow \text{raise } e \end{aligned}$$

Конструкция `case` проверяет, помечено ли возникшее исключение тегом l . Если это так, то она связывает с переменной x значение, переданное с исключением, и выполняет обработчик h . В противном случае управление получает вариант, принятый по умолчанию, который *повторно бро-сает* (re-raises) исключение. Исключение продолжит распространяться (возможно, при этом будучи несколько раз поймано и повторно брошено), пока оно либо не достигнет обработчика, готового с ним работать, либо не окажется на самом верхнем уровне и не приведет к завершению всей программы.

¹Можно пойти еще дальше и вообще сделать расширяемые вариантыкие типы разрешенной языковой конструкцией, но проектировщики ML предпочли просто считать `exp` особым случаем.

²Поскольку форма `exception` связывает идентификатор, мы всегда можем сделать так, чтобы l отличался от уже используемых в T_{exn} тегов, произведя над ним при необходимости альфа-конверсию.

5. В Java вместо расширяемых вариантов для поддержки исключений, определяемых программистом, используются классы. В языке есть встроенный класс `Throwable`; экземпляр `Throwable` или любого из его подклассов можно использовать в конструкции `throw` (аналогично нашему `raise`) или `try ...catch` (аналогично нашему `try ...with`). Ноевые исключения можно объявлять, просто определяя новые подклассы `Throwable`.

Существует тесное соответствие между таким механизмом обработки исключений и тем, который принят в ML. Грубо говоря, объект-исключение в Java во время исполнения представляется тегом, указывающим его класс (что прямо соответствует тегу расширяемого варианта в ML), а также переменными экземпляра класса (что соответствует дополнительной информации, помеченной этим тегом).

Исключения Java в некоторых отношениях идут несколько дальше, чем в ML. Во-первых, существует естественный частичный порядок тегов исключений, определяемый порядком наследования *подклассов* (*subclasses*). Обработчик для исключения 1 будет перехватывать все исключения, имеющие класс 1 или любой из подклассов 1. Кроме того, Java различает *исключения* (exceptions) (подклассы встроенного класса `Exception` — подкласса `Throwable`), которые могут быть перехвачены и исправлены прикладными программами, и *ошибки* (errors) (подклассы класса `Error` — другого подкласса `Throwable`), обозначающие серьезные неполадки, как правило, ведущие к остановке программы. Главное различие между ними состоит в правилах проверки типов, которые требуют, чтобы методы явно объявляли, какие исключения (но *не* ошибки) они могут породить.

Упражнение 14.3.1 [★★★]: Описание расширяемых вариантов типов в варианте номер 4 недостаточно формально. Покажите, как сделать его точным.

Упражнение 14.3.2 [★★★★]: Мы упомянули, что исключения в Java (*те, которые являются подклассами `Exception`*) отслеживаются строже, чем исключения в ML (или *те исключения, что мы определили здесь*): каждое исключение, которое может быть порождено тем или иным методом класса, должно быть указано в типе этого метода. Расширьте свое решение упражнения 14.3.1 так, чтобы тип, указываемый для функции, содержал не только типы ее аргументов и результата, но и множество исключений, которые в ней могут возникнуть. Докажите, что ваша система соблюдает типовую безопасность.

Упражнение 14.3.3 [★★★]: С помощью методов, подобных обсуждаемым в этой главе, можно формально определить многие другие управляющие конструкции. Читатели, знакомые с оператором «вызов с текущим продолжением» (*call-with-current-continuation*, *call/cc*) из языка Scheme (см. Clinger, Friedman, and Wand, 1985; Kelsey, Clinger, and Rees, 1998; Dybvig, 1996; Friedman, Wand, and Haynes, 2001), могут получить удовольствие, по-

пытались сформулировать правила типизации для типа T -продолжений $\text{Cont } T$ — т. е., типа продолжений, ожидающих аргумент типа T .

Часть III

Подтипы

Глава 15

Подтипы

На протяжении нескольких последних глав мы занимались исследованием вопросов типизации разнообразных языковых конструкций в рамках простого типизированного лямбда-исчисления. В этой главе вводится более существенное расширение: *подтипы* (subtyping) (иногда называемое *полиморфизмом через подтипы*, или subtyping polymorphism). Все предыдущие изученные нами конструкции можно было сформулировать более или менее независимо друг от друга. В отличие от них, наличие подтипов требует существенного расширения языка и затрагивает большую часть языковых элементов нетривиальным образом.

Подтипы, как правило, встречаются в *объектно-ориентированных* (object-oriented) языках, и часто рассматриваются как существенный элемент объектно-ориентированного стиля. Эта связь исследуется в главе 18; однако пока что мы описываем подтипы в ограниченном объеме, затрагивая только функции и записи. Уже тут возникает множество интересных вопросов, связанных с наличием подтипов. В §15.5 описывается сочетание подтипов с некоторыми другими конструкциями, которые мы рассматривали в предыдущих главах. В последнем разделе главы (§15.6) мы детальнее исследуем семантику подтипов, в которой их использование соответствует добавлению в исполняемый код *преобразований типа* (coercions).

15.1. Включение

Без подтипов правила простого типизированного лямбда-исчисления могут быть чрезмерно жесткими. Система типов требует, чтобы типы аргументов точно совпадали с типами областей определения функций. Это приводит к тому, что при проверке отвергаются многие программы, которые программисту кажутся очевидно правильными. Вспомним, например, правило для

В этой главе изучается $\lambda_{<:}$, простое типизированное лямбда-исчисление с подтипами (рис. 15.1) и записями (рис. 15.3); соответствующая реализация на OCaml называется `rsubst`. (В некоторых примерах также используются числа; для их проверки требуется `fullsub`.)

применения функций:

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad (\text{T-APP})$$

Вполне корректный терм

$(\lambda \ r : \{x:\text{Nat}\}. \ r.x) \ \{x=0, y=1\}$

не удовлетворяет этому правилу типизации, поскольку тип аргумента — $\{x:\text{Nat}, y:\text{Nat}\}$, тогда как функция принимает $\{x:\text{Nat}\}$. Ясно, однако, что для функции существенно только то, чтобы аргумент был записью с полем x ; имеются ли другие поля и каковы они, ей безразлично. Более того, это требование можно распознать по типу функции — не нужно проверять ее тело и убеждаться в том, что поля кроме x не используются. Передача аргумента типа $\{x:\text{Nat}, y:\text{Nat}\}$ в функцию, ожидающую тип $\{x:\text{Nat}\}$, всегда безопасна.

Цель введения подтипов в язык — уточнить правила типизации так, чтобы такие термы стали разрешенными. Мы добиваемся этого, формализуя интуитивное представление о том, что некоторые типы более «информативны», чем другие: мы говорим, что S является *подтипом* T (subtype) (записывается как $S <: T$), имея в виду, что всякий терм типа S можно безопасно использовать в контексте, в котором ожидается тип T . Такую точку зрения на подтипы часто называют *принципом безопасной подстановки* (principle of safe substitution).

Более простое интуитивное объяснение таково: запись $S <: T$ означает, что «каждое значение, соответствующее типу S , соответствует также типу T », то есть, «элементы S являются подмножеством элементов T ». В §15.6 мы увидим, что иногда полезны другие, более тонкие, интерпретации подтипов, но такая *семантика подмножеств* (subset semantics) достаточна для большинства нужд.

Связь между отношением типизации и нашим новым отношением подтипов обеспечивается новым правилом типизации — так называемым *правилом включения* (subsumption):

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad (\text{T-SUB})$$

Это правило говорит, что если $S <: T$, то всякий элемент S также является элементом T . Например, если мы определим отношение подтипования так, чтобы $\{x:\text{Nat}, y:\text{Nat}\} <: \{x:\text{Nat}\}$, то при помощи правила T-SUB мы сможем вывести $\vdash \{x=0, y=1\} : \{x:\text{Nat}\}$, и таким образом присвоить тип нашему исходному примеру.

15.2. Отношение подтипов

Отношение подтипования формально описывается через набор правил вывода, с помощью которых можно получить утверждения вида $S <: T$, что читается как « S — подтип T » (или « T — надтип S »). Мы рассмотрим по отдельности каждую разновидность типов (типы функций, типы записей и т. д.); для

каждой из них мы формализуем правила, в соответствии с которыми можно разрешить использование элементов одного типа при том, что ожидается другой тип.

Прежде чем приступить к правилам для конкретных разновидностей типов, мы сделаем два общих предположения: во-первых, отношение подтипования должно быть рефлексивным:

$$S \subset: S \quad (\text{S-REFL})$$

а во-вторых, оно должно быть транзитивным:

$$\frac{S \subset: U \quad U \subset: T}{S \subset: T} \quad (\text{S-TRANS})$$

Эти правила прямо следуют из нашего интуитивного представления о безопасности подстановки.

В случае типов записей, как мы уже видели, мы хотим считать тип $S = \{k_1:S_1 \dots k_m:S_m\}$ подтипом $T = \{l_1:T_1 \dots l_n:T_n\}$, если T содержит меньше полей, чем S . В частности, безопасно «забывать» последние поля в типах записей. Эта интуитивная идея отражена в так называемом правиле *подтипование в ширину* (width subtyping):

$$\{l_i:T_i^{i \in 1..n+k}\} \subset: \{l_i:T_i^{i \in 1..n}\} \quad (\text{S-RCDWIDTH})$$

Может показаться странным, что «меньший» тип — подтип — содержит *больше* полей. Это проще всего понять, приняв менее строгую точку зрения на типы записей, чем в §11.8, согласно которой тип $\{x:\text{Nat}\}$ описывает «множество записей, имеющих, по меньшей мере, поле x типа Nat». Элементами этого типа являются не только такие значения как $\{x=3\}$ и $\{x=5\}$, но также и такие значения как $\{x=3, y=100\}$ и $\{x=3, a=true, b=true\}$. Аналогично, тип $\{x:\text{Nat}, y:\text{Nat}\}$ описывает записи, имеющие по крайней мере поля x и y , оба типа Nat. Значениями этого типа являются такие записи, как $\{x=3, y=100\}$ и $\{x=3, y=100, z=true\}$, но не $\{x=3\}$ и не $\{x=3, a=true, b=true\}$. Таким образом, множество значений, принадлежащих ко второму типу, является строгим подмножеством множества значений, принадлежащих к первому типу. Более длинный тип-запись соответствует более жесткой, т. е. более информативной, спецификации; таким образом, он описывает меньшее множество значений.

Правило подтипования в ширину применимо только к типам записей, для которых типы общих полей совпадают. Можно также позволить различаться типам отдельных полей, если типы каждого соответствующего поля являются подтипами. Эта интуитивная концепция выражается правилом *подтипования в глубину* (depth subtyping):

$$\frac{\text{для каждого } i, S_i \subset: T_i}{\{l_i:S_i^{i \in 1..n}\} \subset: \{l_i:T_i^{i \in 1..n}\}} \quad (\text{S-RCDDEPTH})$$

Следующее дерево вывода подтипования использует правила S-RCDWIDTH и S-RCDDEPTH, демонстрируя, что тип вложенных запи-

сей $\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}$ является подтипом $\{x:\{a:Nat\},y:\{\}\}$:

$$\frac{\{a:Nat,b:Nat\} <: \{a:Nat\}}{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\} <: \{x:\{a:Nat\},y:\{\}\}} \text{S-RCDWIDTH} \quad \frac{\{m:Nat\} <: \{\}}{\{x:\{a:Nat\},y:\{\}\}} \text{S-RCDWIDTH} \quad \frac{}{\{x:\{a:Nat\},y:\{\}\}} \text{S-RCDDEPTH}$$

Если мы хотим с помощью S-RCDDEPTH уточнить тип только одного поля записи (а не всех полей, как в предыдущем примере), то для остальных полей можно получить тривиальные отношения подтипования с помощью правила S-REFL:

$$\frac{\{a:Nat,b:Nat\} <: \{a:Nat\} \quad \{m:Nat\} <: \{m:Nat\}}{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\} <: \{x:\{a:Nat\},y:\{m:Nat\}\}} \text{S-RCDWIDTH} \quad \text{S-REFL} \quad \frac{}{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} \text{S-RCDDEPTH}$$

С помощью правила транзитивности S-TRANS можно сочетать подтиповование в ширину и в глубину. Например, можно получить надтип, расширив тип одного поля и отбросив другое:

$$\frac{\{x:\{a:Nat,b:Nat\}, y:\{m:Nat\}\} <: \{x:\{a:Nat,b:Nat\}\} \quad \{a:Nat,b:Nat\} <: \{a:Nat\}}{\{x:\{a:Nat,b:Nat\}, y:\{m:Nat\}\} <: \{x:\{a:Nat\}\}} \text{S-RCDWIDTH} \quad \frac{\{x:\{a:Nat,b:Nat\}\} <: \{x:\{a:Nat\}\}}{\{x:\{a:Nat,b:Nat\}, y:\{m:Nat\}\} <: \{x:\{a:Nat\}\}} \text{S-RCDDEPTH} \quad \text{S-TRANS}$$

Последнее правило для подтиповования записей выводится из наблюдения о том, что порядок полей в записи не оказывает никакого влияния на то, как ее можно безопасно использовать, поскольку единственная операция, которую можно производить с записью после ее создания — проекция ее полей — нечувствительна к порядку полей.

$$\frac{\{k_j:S_j^{j \in 1..n}\} \text{ является перестановкой } \{l_i:T_i^{i \in 1..n}\} \quad \{k_j:S_j^{j \in 1..n}\} <: \{l_i:T_i^{i \in 1..n}\}}{\{x:\{a:Nat,b:Bool,c:Nat\}\} \text{ является подтипом } \{a:Nat,b:Bool,c:Nat\}} \text{S-RCDPERM}$$

Например, согласно правилу S-RCDPERM, тип $\{c:Top,b:Bool,a:Nat\}$ является подтипом $\{a:Nat,b:Bool,c:Top\}$, и наоборот. (Отсюда следует, что отношение подтипования *не будет антисимметрично*.)

Можно использовать S-RCDPERM совместно с S-RCDWIDTH и S-TRANS, чтобы отбрасывать поля в любом месте записи, а не только в ее конце.

Упражнение 15.2.1 [★]: Нарисуйте дерево вывода, показывающее, что $\{x:Nat,y:Nat,z:Nat\}$ является подтипом $\{y:Nat\}$.

Каждое из правил S-RCDWIDTH, S-RCDDEPTH и S-RCDPERM дает нам отдельную степень свободы при работе с записями. В целях обсуждения имеет смысл оставить их в виде трех отдельных правил. Например, существуют языки, в которых разрешена только часть этих правил. Например, большинство

вариантов *исчисления объектов* (object calculus) Абади и Карделли (Abadi and Cardelli, 1996) не используют подтиповование в глубину. Однако при реализации языка полезно сочетать эти правила в одном макроправиле, которое обрабатывает все три вида подтиповования сразу. Это правило обсуждается в следующей главе (ср. с. 237).

Так как мы работаем с языком высшего порядка, где в качестве аргументов в функцию могут передаваться не только числа и записи, но и другие функции, то требуется также придумать правила подтиповирования для функциональных типов — т. е., мы должны указать, в каких случаях безопасно использовать функцию одного типа там, где ожидается функция другого типа.

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{S-ARROW})$$

Заметим, что отношение подтиповирования «срабатывает в обратную сторону» (*контравариантно*, contravariant) для типов аргументов в левой предпосылке, но имеет одно и то же направление (*covariantно*, covariant) для типов результатов и для самих функциональных типов. Интуитивное представление состоит в том, что, имея функцию f типа $S_1 \rightarrow S_2$, мы знаем, что f принимает аргументы типа S_1 ; ясно, что f может также принять аргумент типа T_1 , где T_1 — любой подтип S_1 . Тип f сообщает нам также, что эта функция возвращает элементы типа S_2 ; мы имеем право считать, что они также принадлежат и к T_2 , где T_2 — любой надтип S_2 . То есть, любая функция f типа $S_1 \rightarrow S_2$ может также рассматриваться как функция типа $T_1 \rightarrow T_2$.

Другая возможная точка зрения состоит в том, что безопасно разрешить использование функции типа $S_1 \rightarrow S_2$ в любом контексте, в котором ожидается тип $T_1 \rightarrow T_2$, если ни один аргумент, который может быть передан в функцию, не будет для нее неожиданным ($T_1 <: S_1$), и ни один ее результат не окажется неожиданным для контекста вызова ($S_2 <: T_2$).

Наконец, удобно иметь тип, являющийся надтипом всех остальных типов. Мы вводим новую типовую константу **Top**, а также правило, которое делает **Top** наибольшим элементом в отношении подтиповирования:

$$S <: \text{Top} \quad (\text{S-Top})$$

Тип **Top** подробнее обсуждается в §15.4.

С формальной точки зрения, отношение подтиповирования есть наименьшее отношение, замкнутое относительно установленных нами правил. Для удобства ссылок, на рис. 15.1, 15.2 и 15.3 приведено полное определение простого типизированного лямбда-исчисления с записями и подтипами, причем синтаксические формы и правила, введенные нами в этой главе, выделены серым. Заметим, что присутствие правил рефлексивности и транзитивности означает, что отношение подтиповирования является *предпорядком* (preorder). Однако, из-за правила перестановки полей в записи, частичным порядком оно не является: имеется много пар различных типов, где каждый элемент пары — подтип другого.

Завершая обсуждение отношения подтиповирования, убедимся в том, что теперь мы можем присвоить тип терму-примеру, с которого началась эта глава.

$\rightarrow \text{<: Top}$

Основано на λ_{\rightarrow} (9.1)

Синтаксис

$$\begin{array}{ll} t ::= & \text{термы:} \\ x & \text{переменная} \\ \lambda x:T.t & \text{абстракция} \\ t t & \text{применение} \end{array}$$

$$v ::= \lambda x:T.t \quad \text{значения: значение-абстракция}$$

$$T ::= \begin{array}{ll} \text{типы:} \\ \text{Top} & \text{максимальный тип} \\ T \rightarrow T & \text{тип функций} \end{array}$$

$$\Gamma ::= \begin{array}{ll} \text{контексты:} \\ \emptyset & \text{пустой контекст} \\ \Gamma, x:T & \text{связывание термовой} \\ & \text{переменной} \end{array}$$

Вычисление

$$t \rightarrow t'$$

$$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2} \quad (\text{E-APP2})$$

$$(\lambda x:T_{11}.t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APPABS})$$

Подтипы

$$S \text{ <: } T$$

$$S \text{ <: } S$$

(S-REFL)

$$\frac{S \text{ <: } U \quad U \text{ <: } T}{S \text{ <: } T} \quad (\text{S-TRANS})$$

$$S \text{ <: } \text{Top}$$

(S-TOP)

$$\frac{T_1 \text{ <: } S_1 \quad S_2 \text{ <: } T_2}{S_1 \rightarrow S_2 \text{ <: } T_1 \rightarrow T_2}$$

(S-ARROW)

Типизация

$$\Gamma \vdash t : T$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

$$\frac{\Gamma, x:T_1 \vdash t_2:T_2}{\Gamma \vdash \lambda x:T_1.t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad (\text{T-APP})$$

$$\frac{\Gamma \vdash t : S \quad S \text{ <: } T}{\Gamma \vdash t : T} \quad (\text{T-SUB})$$

Рис. 15.1. Простое типизированное лямбда-исчисление с подтипами ($\lambda_{\text{<:}}$)

Чтобы не выходить за пределы страницы, воспользуемся следующими сокращениями:

$$\begin{array}{ll} f & \stackrel{\text{def}}{=} \lambda x:\{x:\text{Nat}\}. \ r.x \\ xy & \stackrel{\text{def}}{=} \{x=0, y=1\} \end{array} \quad \begin{array}{ll} Rx & \stackrel{\text{def}}{=} \{x:\text{Nat}\} \\ Rxy & \stackrel{\text{def}}{=} \{x:\text{Nat}, y:\text{Nat}\} \end{array}$$

Предполагая обычные правила типизации для числовых констант, мы можем

$\rightarrow \{\}$ Расширяем λ_{\rightarrow} (9.1)

Новые синтаксические формы

$t ::= \dots$	термы:
$\{l_i=t_i^{i \in 1..n}\}$	запись
$t.l$	проекция

$$\frac{t_1 \rightarrow t'_1}{t_1.l \rightarrow t'_1.l} \quad (\text{E-PROJ})$$

$v ::= \dots$	значения:
$\{l_i=v_i^{i \in 1..n}\}$	запись-значение

$$\frac{t_j \rightarrow t'_j}{\{l_i=v_i^{i \in 1..j-1}, l_j=t_j, l_k=t_k^{k \in j+1..n}\} \rightarrow \{l_i=v_i^{i \in 1..j-1}, l_j=t'_j, l_k=t_k^{k \in j+1..n}\}} \quad (\text{E-RCD})$$

$T ::= \dots$	типы:
$\{l_i:T_i^{i \in 1..n}\}$	тип записей

Новые правила типизации $\boxed{\Gamma \vdash t : T}$

Новые правила вычисления

 $t \rightarrow t'$

$$\{l_i=v_i^{i \in 1..n}\}.l_j \rightarrow v_j \quad (\text{E-PROJRCD})$$

$$\frac{\text{для каждого } i, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i=t_i^{i \in 1..n}\} : \{l_i:T_i^{i \in 1..n}\}} \quad (\text{T-RCD})$$

$$\frac{\Gamma \vdash t_1 : \{l_i:T_i^{i \in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \quad (\text{T-PROJ})$$

Рис. 15.2. Записи (совпадает с рис. 11.7)

построить такое дерево вывода для утверждения типизации $\vdash f \ xy : \text{Nat}$:

$$\frac{\vdash 0 : \text{Nat} \quad \vdash 1 : \text{Nat}}{\vdash \text{xy} : \text{Rxy}} \text{ T-RCD} \quad \frac{\text{Rxy} <: \text{Rx}}{\vdash \text{xy} : \text{Rx}} \text{ S-RCDWIDTH} \quad \frac{\vdash \text{xy} : \text{Rx}}{\vdash \text{f} \ xy : \text{Nat}} \text{ T-SUB}$$

$$\vdash \text{f} : \text{Rx} \rightarrow \text{Nat} \quad \frac{}{\vdash \text{f} \ xy : \text{Nat}} \text{ T-APP}$$

Упражнение 15.2.2 [★]: Является ли приведенное дерево вывода для $\vdash f \ xy : \text{Nat}$ единственным?Упражнение 15.2.3 [★]: (1) Сколько различных подтипов имеет $\{a:Top, b:Top\}$? (2) Можно ли найти бесконечную исходящую цепочку в отношении подтипования — то есть, бесконечную последовательность типов S_0, S_1, \dots , такую, что каждый S_{i+1} является подтиповом S_i ? (3) Можно ли найти бесконечную восходящую цепочку?

Упражнение 15.2.4 [★]: Существует ли тип, являющийся подтиповом любого другого типа? Существует ли функциональный тип, являющийся надтиповом любого другого функционального типа?

$\rightarrow \{\} <:$ Расширяет $\lambda_{<:}$ (15.1) и правила для простых записей (15.2)

Новые правила подтиповирования
 $S <: T$

$$\{1_i : T_i^{i \in 1..n+k}\} <: \{1_i : T_i^{i \in 1..n}\} \quad (\text{S-RCDWIDTH})$$

$$\frac{\{k_j : S_j^{j \in 1..n}\} \text{ является перестановкой } \{1_i : T_i^{i \in 1..n}\}}{\{k_j : S_j^{j \in 1..n}\} <: \{1_i : T_i^{i \in 1..n}\}} \quad (\text{S-RCDPERM})$$

$$\frac{\text{для каждого } i, \quad S_i <: T_i}{\{1_i : S_i^{i \in 1..n}\} <: \{1_i : T_i^{i \in 1..n}\}} \quad (\text{S-RCDDEPTH})$$

Рис. 15.3. Записи и подтипы

Упражнение 15.2.5 [★★]: Допустим, что мы расширили наше исчисление конструктором типов-произведений $T_1 \times T_2$, как описано в §11.6. Естественно добавить правило подтиповирования

$$\frac{S_1 <: T_1 \quad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2} \quad (\text{S-PRODDEPTH})$$

соответствующее S-RCDDEPTH для записей. Правильно ли будет добавить также и такое правило:

$$T_1 \times T_2 <: T_1 \quad (\text{S-PRODWIDTH})$$

15.3. Свойства подтипов и типизации

Формально определив лямбда-исчисление с подтипами, мы должны теперь проверить, что это определение ничему не противоречит — в частности, что теоремы о сохранении и продвижении для простого типизированного лямбда-исчисления продолжают выполняться при наличии подтипов.

Упражнение 15.3.1 [РЕКОМЕНДУЕТСЯ, ★★]: Прежде чем читать дальше, попробуйте предугадать, где в доказательствах могут возникнуть сложности. В частности, представим, что мы ошиблись при определении подтиповирования и добавили к имеющимся у нас еще одно, неверное правило подтиповирования. Какие свойства системы могут при этом потеряться? С другой стороны, предположим, что мы отбросили одно из правил подтиповирования — могут ли при этом исчезнуть какие-либо свойства?

Вначале сформулируем одно ключевое свойство отношения подтиповирования — аналог леммы об инверсии для отношения типизации в простом типизированном лямбда-исчислении (лемма 9.3.1). Если нам известно, что S является

подтипов функционального типа, то лемма об инверсии с подтипами говорит, что и сам S должен быть функциональным типом; более того, она утверждает, что типы слева от стрелок должны быть связаны отношением подтиповирования контравариантно, а типы справа от стрелок — ковариантно. Подобные соображения работают и в случае, когда известно, что S является подтиповым типом записей: мы знаем, что S содержит больше полей ($S\text{-RCDWIDTH}$) в произвольном порядке ($S\text{-RCDPERM}$), и типы общих полей связаны отношением подтиповирования ($S\text{-RCDEPTHN}$).

Лемма 15.3.2 [Инверсия отношения подтиповирования]:

1. Если $S <: T_1 \rightarrow T_2$, то S имеет вид $S_1 \rightarrow S_2$, причем $T_1 <: S_1$ и $S_2 <: T_2$.
2. Если $S <: \{l_i : T_i^{i \in 1..n}\}$, то S имеет вид $\{k_j : S_j^{j \in 1..m}\}$, причем содержит по крайней мере метки полей $\{l_i^{i \in 1..n}\}$ — т. е., $\{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\}$, и для каждой общей метки поля $l_i = k_j$, $S_j <: T_i$.

Доказательство: УПРАЖНЕНИЕ [РЕКОМЕНДУЕТСЯ, ★★].

Чтобы доказать, что типы сохраняются при вычислении, начнем с леммы об инверсии отношения типизации (ср. с леммой 9.3.1 для простого типизированного лямбда-исчисления). Вместо того, чтобы сформулировать эту лемму в самой общей форме, мы приводим здесь только те частные случаи, которые необходимы для доказательства теоремы о сохранении. Общую форму можно получить, изучив алгоритмическое отношение типизации, описанное в следующей главе (см. определение 16.2.2).

Лемма 15.3.3

1. Если $\Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2$, то $T_1 <: S_1$, и $\Gamma, x : S_1 \vdash s_2 : T_2$.
2. Если $\Gamma \vdash \{k_a = S_a^{a \in 1..m}\} : \{l_i : T_i^{i \in 1..n}\}$, то $\{l_i^{i \in 1..n}\} \subseteq \{k_a^{a \in 1..m}\}$, и $\Gamma \vdash s_a : T_i$ для каждой общей метки поля $k_a = l_i$.

Доказательство: Несложная индукция по деревьям вывода типов с использованием леммы 15.3.2 для правила T-SUB.

Затем нам потребуется лемма о подстановке для отношения типизации. Формулировка леммы остается той же, что и в случае простого типизированного лямбда-исчисления (лемма 9.3.8), и доказательство также практически совпадает.

Лемма 15.3.4 [Подстановка]: Если $\Gamma, x : S \vdash t : T$ и $\Gamma \vdash s : S$, то $\Gamma \vdash [x \mapsto s]t : T$.

Доказательство: Индукция по деревьям вывода типов. Требуется рассмотреть новые варианты для правил T-SUB и для правил типизации записей T-RCD и T-PROJ, используя очевидным образом предположение индукции. В остальном доказательство такое же, как для 9.3.8.

Теорема о сохранении формулируется точно так же, как и раньше. Однако наличие подтипов усложняет ее доказательство в нескольких местах.

Теорема 15.3.5 [СОХРАНЕНИЕ]: *Если $\Gamma \vdash t : T$ и $t \rightarrow t'$, то $\Gamma \vdash t' : T$.*

Доказательство: Прямолинейная индукция по деревьям вывода типов. Большинство случаев похожи на доказательство сохранения для простого типизированного лямбда-исчисления (9.3.9). Нужно лишь рассмотреть новые варианты для правил типизации записей и для включения.

Вариант T-VAR: $t = x$

Не может возникнуть (не существует правил вычисления для переменных).

Вариант T-ABS: $t = \lambda x:T_1. t_2$

Не может возникнуть (t уже является значением).

Вариант T-APP: $t = t_1 t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \\ T = T_{12}$

Исходя из правил вычисления на рис. 15.1 и 15.2, мы видим, что имеется три правила, позволяющих вывести $t \rightarrow t'$: E-APP1, E-APP2 и E-APPABS. Рассмотрим эти варианты.

Подвариант E-APP1: $t_1 \rightarrow t'_1 \quad t' = t'_1 t_2$

Нужный нам результат следует из предположения индукции и правила T-APP.

Подвариант E-APP2: $t_1 = v_1 \quad t_2 \rightarrow t'_2 \quad t' = v_1 t'_2$

Аналогично.

Подвариант E-APPABS: $t_1 = \lambda x:S_{11}. t_{12} \quad t_2 = v_2 \quad t' =$

$[x \mapsto v_2] t_{12}$

По лемме 15.3.3(1), $T_{11} <: S_{11}$ и $\Gamma, x:S_{11} \vdash t_{12} : T_{12}$. Согласно правилу T-SUB, $\Gamma \vdash t_2 : S_{11}$. Отсюда, используя лемму о подстановке (15.3.4), получаем $\Gamma \vdash t' : T_{12}$.

Вариант T-RCD: $t = \{l_i = t_i^{i \in 1..n}\} \quad \Gamma \vdash t_i : T_i$ для каждого i
 $T = \{l_i : T_i^{i \in 1..n}\}$

Единственное правило вычисления, содержащее в левой части запись, — это E-RCD. Исходя из предпосылки этого правила, мы видим, что $t_j \rightarrow t'_j$ для некоторого поля t_j . Требуемый результат следует из предположения индукции (примененного к соответствующей предпосылке $\Gamma \vdash t_j : T_j$) и правила T-RCD.

Вариант T-PROJ: $t = t_1 . l_j \quad \Gamma \vdash t_1 : \{l_i : T_i^{i \in 1..n}\} \quad T = T_j$

Исходя из правил вычисления на рис. 15.1 и 15.2, мы видим, что $t \rightarrow t'$ может быть выведено из двух правил: E-PROJ и E-PROJRCD.

Подвариант E-PROJ: $t_1 \rightarrow t'_1 \quad t' = t'_1 . l_j$

Результат следует из предположения индукции и правила T-PROJ.

Подвариант E-PROJRCD: $t_1 = \{k_a = v_a^{a \in 1..m}\} \quad l_j = k_b \quad t' = v_b$

По лемме 15.3.3(2), имеем $\{l_i^{i \in 1..n}\} \subseteq \{k_a^{a \in 1..m}\}$ и $\Gamma \vdash v_a : T_i$ для каждого $k_a = l_i$. В частности, $\Gamma \vdash v_b : T_j$, как нам и требуетсяся.

Вариант T-SUB: $t : S \quad S <: T$

Согласно предположению индукции, $\Gamma \vdash t' : S$. По правилу T-SUB, $\Gamma \vdash t : T$.

Чтобы показать, что правильно типизированные термы никогда не оказываются в тупике, мы (как и в главе 9) сначала докажем лемму о канонических формах, которая фиксирует возможные виды термов-значений, принадлежащих функциональным типам и типам записей.

Лемма 15.3.6 [Канонические формы]:

1. Если v — замкнутое значение типа $T_1 \rightarrow T_2$, то v имеет вид $\lambda x:S_1 . t_2$.
2. Если v — замкнутое значение типа $\{l_i:T_i^{i \in 1..n}\}$, то v имеет вид $\{k_a=v_a^{a \in 1..m}\}$, причем $\{l_i^{i \in 1..n}\} \subseteq \{k_a^{a \in 1..m}\}$.

Доказательство: УПРАЖНЕНИЕ [РЕКОМЕНДУЕТСЯ, ★★★].

Утверждение теоремы о продвижении и её доказательство достаточно близки к тому, что мы уже видели для простого типизированного лямбда-исчисления. Большинство трудностей, связанных с подтипаами, выведено в лемму о канонических формах, и теперь требуется внести только несколько небольших изменений.

Теорема 15.3.7 [ПРОДВИЖЕНИЕ]: *Если t — замкнутый правильно типизированный терм, то либо t является значением, либо существует некоторый терм t' такой, что $t \rightarrow t'$.*

Доказательство: Прямолинейная индукция по деревьям вывода типов. Вариант с переменной возникнуть не может (поскольку t замкнут). В варианте с лямбда-абстракцией искомый результат следует немедленно, так как абстракции являются значениями. Оставшиеся варианты более интересны.

Вариант T-APP: $t = t_1 \ t_2 \quad \vdash t_1 : T_{11} \rightarrow T_{12} \quad \vdash t_2 : T_{11} \quad T = T_{12}$
Согласно предположению индукции, терм t_1 либо является значением, либо способен проделать шаг вычисления. То же самое верно и по отношению к t_2 . Если t_1 способен проделать шаг, то применимо правило T-APP1. Если t_1 — значение, а t_2 способен проделать шаг, то применимо правило T-APP2. Наконец, если оба терма t_1 и t_2 являются значениями, то, согласно лемме о канонических формах (15.3.6), t_1 имеет вид $\lambda x:S_{11} . t_{12}$, так что к t применимо правило E-APPABs.

Вариант T-RCD: $t = \{l_i=t_i^{i \in 1..n}\} \quad \text{для каждого } i \in 1..n, \vdash t_i : T_i \quad T = \{l_i : T_i^{i \in 1..n}\}$

Согласно предположению индукции, каждый из термов t_i либо является значением, либо способен проделать шаг вычисления. Если все эти термы — значения, то t сам является значением. Если же какой-либо из них способен проделать шаг, то к t применимо правило E-RCD.

Вариант T-PROJ: $t = t_1 . l_j \quad \vdash t_1 : \{l_i:T_i^{i \in 1..n}\} \quad T = T_j$

Согласно предположению индукции, терм t_1 либо является значением, либо способен проделать шаг вычисления. Если t_1 может проделать шаг,

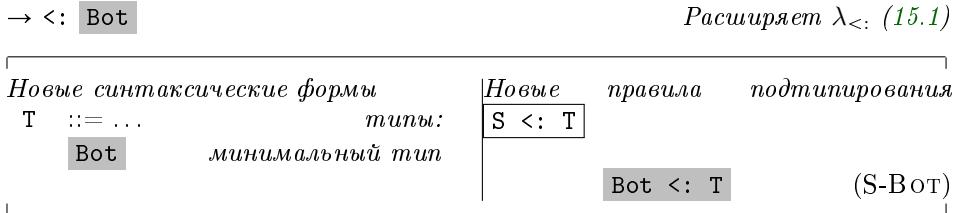


Рис. 15.4. Наименьший тип

то, по правилу E-PROJ, может и t . Если же t_1 — значение, то, согласно лемме о канонических формах (15.3.6), t_1 имеет вид $\{k_a=v_a^{a \in 1..m}\}$, причем $\{l_i^{i \in 1..n}\} \subseteq \{k_a^{a \in 1..m}\}$ и для каждого $l_i = k_j$ имеем $\vdash v_j : T_i$. В частности, l_j входит в набор меток $\{k_a^{a \in 1..m}\}$ терма t_1 , откуда, по правилу E-PROJRCD, мы видим, что и сам терм t способен проделать шаг вычисления.

Вариант T-SUB: $\Gamma \vdash t : S \quad S <: T$

Результат прямо следует из предположения индукции.

15.4. Типы *Top* и *Bottom*

Наибольший тип *Top* не является необходимым элементом простого типизированного лямбда-исчисления с подтипами; его можно убрать, не нарушая базовых свойств системы. Однако большинство описаний включают этот тип, по некоторым причинам. Во-первых, он соответствует типу *Объект*, имеющемуся в большинстве объектно-ориентированных языков программирования. Во-вторых, *Top* служит удобным техническим инструментом в более сложных системах, в которых подтипы сочетаются с полиморфизмом. Например, в Системе $F_{<:}$ (главы 26 и 28) наличие типа *Top* позволяет получить обыкновенную неограниченную квантификацию на основе ограниченной квантификации и, таким образом, упрощает всю систему. Более того, в $F_{<:}$ можно закодировать даже записи и, таким образом, еще больше упростить повествование (по крайней мере, с академической точки зрения); такое кодирование невозможно без *Top*. Наконец, поскольку поведение *Top* устроено несложно, а использование этого типа в примерах часто бывает полезно, нет особых причин от него отказываться.

Естественно спросить, можно ли также дополнить отношение подтипования *наименьшим* элементом — типом *Bot*, который является подтипом всякого другого типа. Это возможно: такое расширение формально представлено на рис. 15.4

Прежде всего, следует заметить, что тип *Bot* пуст; замкнутых значений типа *Bot* не существует. Если бы такое значение, например, v , существовало, то правило включения вместе с правилом *S-Bot* позволили бы нам вывести $\vdash v : \text{Top} \rightarrow \text{Top}$, откуда, по лемме о канонических формах (15.3.6, которая по-прежнему выполняется в нашем расширении), следовало бы, что терм v должен иметь вид $\lambda x:S_1.t_2$ для некоторых S_1 и t_2 . С другой стороны, по пра-

вилу включения, мы также имеем $\vdash v : \{\}$, откуда по лемме о канонических формах следует, что терм *v* должен быть записью. Но из правил синтаксиса ясно, что *v* не может одновременно являться функцией и записью, так что предположение $\vdash v : \text{Bot}$ ведет к противоречию.

Пустота типа *Bot* не означает его бесполезности. Напротив: *Bot* дает программисту удобный способ выразить информацию о том, что некоторые операции (в частности, порождение исключения или вызов продолжения) не должны возвращать никакого значения. Если мы присвоим таким выражениям тип *Bot*, то получим нам два полезных результата: во-первых, это сообщает программисту, что никакого результата не ожидается (поскольку если бы выражение *имело* результат, это оказалось бы значение типа *Bot*); во-вторых, это дает знать программе проверки типов, что такое выражение можно безопасно использовать в контексте, ожидающем любой тип значения. Например, если терм *error* из главы 14,зывающий исключение, получит тип *Bot*, то терм вроде

```

$$\lambda x:T.$$

  if <проверка, что x имеет разумное значение> then
    <вычислить результат>
  else
    error
```

будет правильно типизирован, поскольку, независимо от того, каков тип нормального результата, терм *error* может получить тот же самый тип через включение, так что две ветви условного выражения совместимы, как того требует правило T-If.¹

К сожалению, наличие *Bot* существенно усложняет разработку программы проверки типов для нашей системы. Простой алгоритм проверки типов в языке с подтиปами должен опираться на рассуждения вроде «если терм-применение $t_1 t_2$ правильно типизирован, то t_1 должен иметь функциональный тип». При наличии *Bot* приходится уточнить это утверждение до следующего вида: «если $t_1 t_2$ правильно типизирован, то t_1 должен иметь либо функциональный тип, либо *Bot*»; этот вопрос подробнее обсуждается в §16.4. Сложности еще более возрастают в системах с ограниченной квантификацией; см. §28.8.

Эти сложности показывают, что добавление в систему *Bot* — более серьезный шаг, чем добавление *Top*. В системах, рассматриваемых в этой книге, мы его опускаем.

¹ В языках с полиморфизмом, например, в ML, в качестве типа результата для *error* и других подобных конструкций также можно использовать тип $\forall X.X$. При этом тот же самый результат достигается различными средствами: вместо того, чтобы присваивать *error* тип, который может быть *повышен* до любого другого, ему присваивают схему типа, которая может быть *конкретизирована* до любого типа. Хотя эти два решения основаны на различных механизмах, они в то же время весьма похожи: в частности, тип $\forall X.X$ также пуст.

15.5. Подтипы и другие элементы языка

По мере того, как мы вводим подтипы в наше простое исчисление, продвигая его в направлении полноценных языков программирования, каждый новый элемент языка необходимо тщательно исследовать, чтобы выяснить, как он взаимодействует с подтипами. В этом разделе мы рассмотрим некоторые конструкции, введенные нами на этот момент.² В последующих главах будет обсуждаться (значительно более сложное) взаимодействие подтипов с такими элементами языка, как параметрический полиморфизм (главы 26 и 28), рекурсивные типы (главы 20 и 21) и операторы типов (глава 31).

Приписывание и приведение типов

Оператор *приписывания типа* (*ascription*) $t \text{ as } T$ был введен в §11.4. Он позволяет программисту записать в тексте программы автоматически проверяемое утверждение о том, что некоторый подтерм составного выражения имеет определенный тип. Приписывание типов также используется в примерах в тексте этой книги для управления распечаткой типов — с его помощью программы проверки типов получают указания, что следует использовать более легкую для чтения сокращенную форму, а не тот тип, который был вычислен для терма во внутреннем представлении.

В языках с подтипами, таких, как Java или C++, приписывание типов оказывается намного более интересной операцией. В этих языках оно часто называется *приведением типов* (*casting*) и записывается в виде $(T)t$. Имеются две совершенно различные формы приведения типов — *восходящее* (*up-casts*) и *нисходящее* (*down-casts*). Первая разновидность не представляет сложностей; вторая, включающая *динамическую проверку типов* (*dynamic type-testing*), требует существенного расширения языка.

Восходящее приведение, в котором терму приписывается *надтип* (*supertype*) того типа, который был бы ему присвоен естественным образом, является частным случаем обычного оператора приписывания типа. Мы сообщаем компилятору терм t и тип T , с точки зрения которого мы хотим «рассматривать» t . Программа проверки типов убеждается, что T — действительно один из типов терма t , пытаясь построить дерево вывода

$$\frac{\vdots \quad \vdots}{\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \text{ T-SUB}} \text{ T-ASCRIBE}$$

с использованием «естественного» типа T , правила включения T-SUB и правила приписывания типа из §11.4:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \text{ (T-ASCRIBE)}$$

²Большинство расширений, обсуждаемых в этом разделе, *не реализованы* в программе проверки типов *fullsub*.

Восходящее приведение типов можно рассматривать как разновидность *абстрагирования* (abstraction) — т. е. как способ спрятать некоторые детали значения, чтобы их нельзя было использовать в некотором окружающем контексте. Например, если t является записью (или, в более общем случае, объектом), то при помощи восходящего приведения можно скрыть некоторые из ее полей (или методов).

С другой стороны, нисходящее приведение позволяет приписывать термам типы, которые программа проверки статически вывести не может. Чтобы разрешить нисходящее приведение, мы вносим несколько неожиданное изменение в правило типизации для `as`:

$$\frac{\Gamma \vdash t_1 : S}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-DOWNCASE})$$

То есть, мы проверяем, что терм t_1 правильно типизирован (т. е., имеет некоторый тип S), а затем присваиваем ему тип T , не выдвигая при этом *никаких* требований о взаимном отношении S и T . Например, с помощью нисходящего приведения мы можем написать функцию, принимающую какой угодно вообще аргумент, приводящую этот аргумент к типу записи, содержащей числовое поле, и возвращающую это поле

```
f = λ (x:Top) (x as {a:Nat}).a;
```

В сущности, программист говорит программе проверки типов: «Я знаю (из соображений, слишком сложных, чтобы объяснить их в рамках правил типизации), что f всегда будет применяться к аргументам-записям, содержащим числовое поле a ; я хочу, чтобы ты мне поверила».

Разумеется, слепая вера в такие утверждения имела бы ужасающие последствия для безопасности нашего языка: если программист почему-либо допустит ошибку и применит f к записи, не содержащей поля a , результат может (в зависимости от устройства компилятора) быть совершенно непредсказуемым. Вместо этого, мы должны руководствоваться принципом «Доверяй, но проверяй». Во время компиляции программа проверки просто принимает тип, указанный в нисходящем приведении. Однако она вставляет в код проверку, которая во время выполнения должна убедиться, что представленное значение на самом деле имеет нужный тип. Другими словами, правило вычисления для выражений приписывания типа не должно просто отбрасывать аннотацию, как делало наше исходное правило:

$$v_1 \text{ as } T \rightarrow v_1 \quad (\text{E-ASCRIBE})$$

а вместо этого должно сравнивать реально наблюдаемый (во время выполнения) тип значения с объявленным типом:

$$\frac{\vdash v_1 : T}{v_1 \text{ as } T \rightarrow v_1} \quad (\text{E-DOWNCASE})$$

Например, если мы применяем вышеуказанную функцию f к аргументу `{a=5,b=true}`, то наше правило (успешно) проверит, что $\vdash \{a=5,b=true\} : \{a:Nat\}$. С другой стороны, если мы ее применим к

`{b=true}`, то правило E-DOWNCASE будет непригодно, и вычисление окажется в тупике. Такая проверка во время выполнения восстанавливает свойство сохранения типов.

Упражнение 15.5.1 [★★ →]: Докажите это.

Разумеется, при этом мы теряем свойство продвижения, поскольку правильно типизированная программа может оказаться в тупике при попытке выполнения неверного нисходящего приведения типа. Языки, содержащие нисходящее приведение, обычно решают это одним из двух способов: либо неудачное нисходящее приведение порождает динамическое исключение, которое программа может перехватить и обработать (см. главу 14), либо вместо операции нисходящего приведения используется динамическая проверка типа:

$$\frac{\Gamma \vdash t_1 : S \quad \Gamma, x:T \vdash t_2 : U \quad \Gamma \vdash t_3 : U}{\Gamma \vdash \text{if } t_1 \text{ in } T \text{ then } x \rightarrow t_2 \text{ else } t_3 : U} \quad (\text{T-TYPETEST})$$

$$\frac{\vdash v_1 : T}{\text{if } v_1 \text{ in } T \text{ then } x \rightarrow t_2 \text{ else } t_3 \rightarrow [x \mapsto v_1]t_2} \quad (\text{E-TYPETEST1})$$

$$\frac{\nexists v_1 : T}{\text{if } v_1 \text{ in } T \text{ then } x \rightarrow t_2 \text{ else } t_3 \rightarrow t_3} \quad (\text{E-TYPETEST2})$$

В таких языках как Java нисходящие приведения типов встречаются довольно часто. Например, с их помощью организуется своего рода «полиморфизм для бедных». Например, «классы-контейнеры», такие как `Set` или `List`, в Java мономорфны: вместо того, чтобы иметь тип `List T` (список, содержащий элементы типа `T`) для каждого типа `T`, Java дает программисту только `List` — тип списков, чьи элементы принадлежат наибольшему типу `Object`. Поскольку в Java `Object` является надтипом любого типа объектов, это означает, что на самом деле списки могут содержать что угодно: когда нам требуется добавить какой-то элемент к списку, мы просто используем включение и повышаем тип этого элемента до `Object`. Однако когда мы извлекаем элемент из списка, процедура проверки типов знает только то, что он обладает типом `Object`. Этот тип не позволяет вызывать большинство методов объекта, поскольку описание `Object` упоминает только несколько самых общих методов, имеющихся у всех объектов Java. Чтобы сделать с элементом что-либо полезное, приходится приводить его вниз к некоторому ожидаемому типу `T`.

Многократно предлагалось — например, разработчиками языков Pizza (Odersky and Wadler, 1997), GJ (Bracha, Odersky, Stoutamire, and Wadler, 1998), PolyJ (Myers, Bank, and Liskov, 1997) и NextGen (Cartwright and Steele, 1998), — ввести в систему типов Java настоящий полиморфизм (ср. главу 23), поскольку он и безопасней, и эффективней приема с нисходящим приведением типов, а кроме того, не требует никаких проверок во время исполнения. С другой стороны, такие расширения увеличивают сложность языка, и без того достаточно громоздкого, и нетривиально взаимодействуют с другими элементами языка (см., например, работы Игараси, Пирса и Уодлера (Igarashi, Pierce,

and Wadler, 1999, 2001)); это усиливает позиции тех, кто считает нисходящие приведения разумным практическим компромиссом между безопасностью и сложностью.

Нисходящие приведения играют ключевую роль также и в поддержке *рефлексии* (reflection) в Java. Используя рефлексию, программист может потребовать от среды исполнения Java динамически загрузить файл с байт-кодом и сконструировать экземпляр содержащегося в этом файле класса. Ясно, что процедура проверки типов никак не может статически предсказать, какой класс будет загружены таким образом (к примеру, байт-код может быть получен откуда-нибудь из сети), так что она может только присвоить новым экземплярам наибольший тип `Object`. Опять же, чтобы осуществить какую-либо полезную работу, приходится приводить новый объект к некоторому ожидаемому типу T , обрабатывать исключения, которые могут возникнуть, если класс в байт-коде на самом деле не совместим с этим типом, и, наконец, использовать этот объект с типом T .

В заключение разговора про нисходящие приведения нужно сделать замечание о реализации. Исходя из данных нами правил, кажется, что их введение в язык требует добавления всех механизмов проверки типов в среду времени выполнения. Хуже того: поскольку, как правило, во время выполнения значения представляются иначе, чем во время компиляции (в частности, функции компилируются в байт-код или в команды машинного языка), нам, видимо, придется написать *отдельную* процедуру для вычисления типов, производимого при проверке во время выполнения. Чтобы избежать этого, в реальных языках нисходящие приведения сочетаются с *тегами типов* (type tags) — тегами размером в одно слово (довольно похожими на конструкторы типов данных в ML и на теги вариантов в §11.10), которые предоставляют «метку» информации о типах, используемых при компиляции, и служат для динамической проверки подтипов. В главе 19 подробно рассматривается пример такого механизма.

Варианты

Правила подтиповирования для вариантов (см. §11.10) почти совпадают с правилами для записей; единственное различие состоит в том, что правило подтиповирования в ширину, S-VARIANTWIDTH, позволяет *добавлять*, а не отбрасывать, новые варианты при переходе от подтипа к надтипу. По интуитивному представлению, выражение с тегом $\langle l=t \rangle$ принадлежит к типу вариантов $\langle l_1:T_1 \dots l_n:T_n \rangle$, если его метка l — одна из возможных меток $\{l_i\}$, перечисленных в определении типа; добавление новых меток к этому множеству уменьшает информацию, которой мы располагаем об элементах типа. Одновариантный тип $\langle l_1:T_1 \rangle$ точно указывает, каким меткам обозначаются его элементы; двухвариантный тип $\langle l_1:T_1, l_2:T_2 \rangle$ говорит, что его элементы отмечены либо меткой l_1 , либо меткой l_2 , и т. д. С другой стороны, когда мы *используем* вариантовые значения, это всегда происходит с помощью выражения `case`, в котором на каждый вариант, указанный для типа, имеется одна ветвь. Если вариантов будет больше, это приведет только к тому, что предложения `case` будут включать ненужные дополнительные ветви.

$\rightarrow \langle \rangle < :$ Расширяет $\lambda_{<}$: (15.1) и простые правила для вариантов (11.11)

Новые синтаксические формы

$t ::= \dots$ термы:
 $\langle l=t \rangle$ (без as) постановка
 тега

Новые правила типизации $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle} \quad (\text{T-VARIANT})$$

Новые правила подтипов

$S <: T$

$$\langle l_i : T_i^{i \in 1..n} \rangle <: \langle l_i : T_i^{i \in 1..n+k} \rangle \quad (\text{S-VARIANTWIDTH})$$

$$\frac{\text{для каждого } i, S_i <: T_i}{\langle l_i : S_i^{i \in 1..n} \rangle <: \langle l_i : T_i^{i \in 1..n} \rangle} \quad (\text{S-VARIANTDEPTH})$$

$$\frac{\langle k_j : S_j^{j \in 1..n} \rangle \text{ явл. перестановкой } \langle l_i : T_i^{i \in 1..n} \rangle}{\langle k_j : S_j^{j \in 1..n} \rangle <: \langle l_i : T_i^{i \in 1..n} \rangle} \quad (\text{S-VARIANTPERM})$$

Рис. 15.5. Варианты и подтипы

Еще одно следствие сочетания подтипов и вариантов — то, что мы теперь можем отказаться от аннотаций в конструкции навешивания тега, и писать просто $\langle l=t \rangle$ вместо $\langle l=t \rangle \text{ as } \langle l_i : T_i^{i \in 1..n} \rangle$, как нам приходилось делать в §11.10. Теперь мы можем упростить правило типизации для этой конструкции и сказать, что $\langle l_1 = t_1 \rangle$ имеет точный тип $\langle l_1 : T_1 \rangle$. Любой более крупный вариантный тип получается через включение и правило S-VARIANTWIDTH.

Списки

Мы уже видели несколько примеров ковариантных конструкторов типов (записи и варианты, а также функциональные типы по правой части типа) и один контравариантный конструктор (стрелка, по левой части типа). Конструктор `List` также ковариантен: если у нас есть список, элементы которого имеют тип S_1 , и при этом $S_1 <: T_1$, то мы можем без опаски считать, что список имеет элементы типа T_1 .

$$\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1} \quad (\text{S-LIST})$$

Ссылки

Не все конструкторы типов являются либо ковариантными, либо контравариантными. К примеру, конструктор `Ref` должен быть *инвариантным*, чтобы

сохранить типовую безопасность.

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \quad (\text{S-REF})$$

Чтобы `Ref S1` считался подтиповом `Ref T1`, мы требуем, чтобы `S1` и `T1` были эквивалентны (equivalent) с точки зрения отношения подтипования — каждый из них должен быть подтиповом другого. Это позволяет нам переупорядочивать поля записей внутри конструктора `Ref` — скажем, `Ref {a:Bool,b:Nat} <: Ref {b:Nat,a:Bool}`, — но ничего больше.

Правило подтипования так сильно ограничено потому, что в любом данном контексте значение типа `Ref T1` может использоваться двумя способами: для чтения (оператор `!`) и записи (`:=`). Когда значение используется для чтения, контекст ожидает получить значение типа `T1`. Поэтому, если на самом деле ссылка вернет значение типа `S1`, нужно, чтобы было `S1 <: T1`, а иначе ожидания контекста будут нарушены. С другой стороны, если та же самая ссылочная ячейка используется для записи, то новое значение, предоставленное контекстом, будет иметь тип `T1`. Если тип ссылки на самом деле `Ref S1`, то кто-нибудь потом может прочитать значение по ссылке и использовать его как `S1`; это будет безопасно только в том случае, если `T1 <: S1`.

Упражнение 15.5.2 [★ →] (1) Напишите короткую программу, которая приведет к ошибке времени выполнения (т. е., вычисление зайдет в тупик), если мы откажемся от первой предпосылки S-REF. (2) Напишите другую программу, которая приведет к ошибке, если будет отброшена вторая предпосылка.

Массивы

Ясно, что причины, побудившие нас принять инвариантное правило подтипования для ссылок, действуют и в случае с массивами, поскольку операции с ними включают как разыменование, так и присваивание.

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Array } S_1 <: \text{Array } T_1} \quad (\text{S-ARRAY})$$

Интересно, что Java разрешает ковариантное подтиповование для массивов:

$$\frac{S_1 <: T_1}{\text{Array } S_1 <: \text{Array } T_1} \quad (\text{S-ARRAYJAVA})$$

(в синтаксисе Java, `S1[] <: T1[]`). Такая конструкция исходно была введена, чтобы компенсировать отсутствие параметрического полиморфизма при типизации некоторых базовых операций, вроде копирования массивов. Сейчас большинство специалистов считают ее ошибкой в проектировании языка, поскольку она серьезно снижает производительность программ, использующих массивы. Это происходит потому, что небезопасное правило подтиповования приходится компенсировать проверкой во время выполнения, при *каждом* присваивании *каждому* массиву, чтобы убедиться в том, что записываемое значение принадлежит подтипу действительного типа элементов массива.

Упражнение 15.5.3 [★ ★ ★ →]: Напишите короткую программу на Java с использованием массивов, так, чтобы она проходила проверку типов, но ломалась (порождая *ArrayStoreException*) во время выполнения.

Снова ссылки

Путем введения двух независимых конструкторов типов, **Source** («кран»*) и **Sink** («слив»), можно получить более тонкую реализацию ссылок, впервые исследованную Рейнольдсом (Reynolds, 1988) в языке Forsythe. С интуитивной точки зрения, **Source** T предоставляет возможность считывать значения типа T из ячейки (но не присваивать значение), а **Sink** T предоставляет возможность записывать значение в ячейку. **Ref** T сочетает обе эти возможности и дает разрешение как на чтение, так и на запись.

Правила типизации для разыменования и присваивания (рис. 13.1) модифицируются так, чтобы проверять только соответствующую каждому из них возможность:

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Source } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \quad (\text{T-DEREF})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Sink } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}} \quad (\text{T-ASSIGN})$$

Теперь, если у нас есть только возможность считывать значение ячейки, и гарантируется, что эти значения будут иметь тип S₁, то мы можем без опаски «сузить» эту возможность до возможности читать значения типа T₁, если S₁ является подтипом T₁. Таким образом, конструктор **Source** ковариантен.

$$\frac{S_1 <: T_1}{\text{Source } S_1 <: \text{Source } T_1} \quad (\text{S-SOURCE})$$

С другой стороны, возможность записывать значения типа S₁ в данную ячейку может быть понижена до возможности записывать значения некоторого *меньшего* типа T₁: конструктор **Sink** контравариантен.

$$\frac{T_1 <: S_1}{\text{Sink } S_1 <: \text{Sink } T_1} \quad (\text{S-SINK})$$

Наконец, выражим интуитивное представление о том, что тип **Ref** T₁ представляет комбинацию возможностей к чтению и к записи, при помощи двух правил подтиповирования, которые позволяют сузить ссылку **Ref** до «крана» **Source** или до «слива» **Sink**.

$$\text{Ref } T_1 <: \text{Source } T_1 \quad (\text{S-REFSOURCE})$$

$$\text{Ref } T_1 <: \text{Sink } T_1 \quad (\text{S-REFSINK})$$

*Буквально «источник». Здесь используется метафора входящей и выходящей труб в бассейне. — прим. перев.

Каналы

На таком же интуитивном представлении (и на тех же самых правилах подтиpирования) основан подход к *тиpам каналов* (channel types) в современных языках для параллельного программирования, таких как Pict (Pierce and Turner, 2000; Pierce and Sangiorgi, 1993). Главный тезис здесь состоит в том, что с точки зрения типизации канал межпроцессного взаимодействия ведет себя точно так же, как ссылочная ячейка: его можно использовать для чтения и для записи, и поскольку трудно статически определить, какие операции чтения соответствуют каким операциям записи, единственный простой способ обеспечить типовую безопасность — требовать, чтобы все значения, передаваемые по каналу, принадлежали к одному типу. Теперь при передаче какому-либо процессу только возможности записи в определенный канал, такой процесс может безопасно передавать эту возможность другому процессу, который обещает записывать в канал значения меньшего типа — конструктор «канал для записи» контравариантен. Аналогично, при передаче только возможности чтения из канала её можно безопасно сузить до возможности чтения значений любого большего типа — конструктор «канал для чтения» является ковариантным.

Базовые типы

В полноценном языке с богатым набором базовых типов часто бывает полезно ввести отношение подтиpирования между этими базовыми типами. Например, во многих языках булевские значения `true` и `false` на самом деле представляются в виде чисел 1 и 0. Мы можем, если пожелаем, сообщить об этом программисту, введя аксиому подтиpирования `Bool <: Nat`. Тогда можно писать компактные выражения вроде `5 * b` вместо `if b then 5 else 0`.

15.6. Семантика подтипов, основанная на преобразованиях типов

На протяжении этой главы мы следовали интуитивному представлению о том, что подтипы «не влияют на семантику». Наличие подтипов никак не влияет на процесс исполнения программ; оно лишь позволяет добавить некоторую гибкость в правила типизации. Такая интерпретация проста и естественна, но ведет к некоторому снижению эффективности, особенно при численных расчетах и при доступе к полям записей. Такие потери могут оказаться неприемлемыми для высокопроизводительных реализаций. В этом разделе мы сделаем набросок альтернативной *семантики на основе преобразования типов* (coercion semantics) и обсудим некоторые вопросы, возникающие в связи с этим новым подходом. При желании этот раздел можно пропустить.

Проблемы семантики, основанной на подмножествах

Как мы видели в §15.5, часто бывает удобно разрешить подтиpирование между различными базовыми типами. Однако некоторые «интуитив-

но понятные» связи между базовыми типами могут отрицательно повлиять на производительность. Допустим, например, что мы введем аксиому `Int <: Float`, чтобы можно было использовать целые числа в вычислениях с плавающей точкой без явных преобразований — например, писать `4.5 + 6` вместо `4.5 + intToFloat(6)`. При семантике, основанной на подмножествах, это означает, что множество целых числовых значений должно буквально являться подмножеством значений с плавающей точкой. Однако в большинстве реальных машин конкретные представления целых чисел и чисел с плавающей точкой совершенно различны: целые обычно представлены в формате с дополнением до двух, а числа с плавающей точкой состоят из мантиссы, показателя степени и знака, плюс некоторые особые случаи вроде `NaN` (не-число).

Чтобы согласовать такое расхождение в представлениях и семантику на основе подмножеств, мы можем принять общее *теговое* (*tagged*) (или *упакованное*, *boxed*) представление для чисел: целое число представляется как машинное целое, снабженное тегом (располагающимся либо в отдельном слово-заголовке, либо в старших разрядах слова, содержащего само целое число), а число с плавающей точкой представляется как машинное число с плавающей точкой, снабженное другим тегом. В таком случае тип `Float` относится ко всему множеству помеченных тегами чисел, а тип `Int` относится только к помеченным целым.

Такая схема вполне разумна: она соответствует стратегии представления, реально применяемой во многих современных реализациях языков, в которых теговые биты (или слова) нужны также при сборке мусора. Недостаток ее состоит в том, что все элементарные операции над числами нужно реализовывать как последовательность из машинных команд проверки тега аргументов, команд для извлечения самого числа, одной команды для самой операции, и пары команд для навешивания тега на результат. Изощренные оптимизации в компиляторе могут устранить часть лишних действий, но даже с лучшими из имеющихся методов производительность сильно страдает, особенно в коде, содержащем многочисленной работы, вроде графических или научных вычислений.

Другая проблема с производительностью возникает, когда подтипы сочетаются с записями, в частности, при применении правила перестановки полей. Наше простое правило вычисления для проекции поля

$$\{l_i = v_i^{i \in 1..n}\}.l_j \rightarrow v_j \quad (\text{E-ProjRCD})$$

можно читать так: «найти среди меток полей записи l_j и вернуть соответствующее значение v_j ». Однако в настоящей реализации мы, безусловно, не хотим проводить линейный просмотр полей записи в поисках нужной метки. В языке без подтипов (или с подтиповыми, но без правила перестановки) можно получить значение намного быстрее: если метка l_j идет третьей в *типе* записи, то мы уже во время компиляции знаем, что все значения данного типа будут содержать в третьем поле l_j , так что во время выполнения нам вовсе не нужно смотреть на метки (мы вообще можем исключить их из представления записей во время выполнения, в сущности, превращая записи в кортежи). Чтобы получить значение поля l_j , мы генерируем команду косвенной загрузки через регистр, указывающий на начало записи, со смещением в 3 слова.

Наличие правила перестановки делает этот метод невозможным, поскольку информация о том, что некоторое значение записи принадлежит типу, в котором l_j является третьим полем, ничего не говорит нам о том, где на самом деле внутри записи хранится поле l_j . Опять же, сложные приемы оптимизации и кодирования могут уменьшить потери, но в общем случае проекция поля может так или иначе потребовать поиска во время выполнения.³

Семантика, основанная на преобразовании типов

Обе эти проблемы можно решить, если принять другую семантику, в которой подтиповирование «исчезает при компиляции» и заменяется преобразованиями типа во время выполнения. Если, скажем, во время проверки типов `Int` повышается до `Float`, то во время выполнения мы физически изменяем представление числа с машинного целого на машинное число с плавающей точкой. Аналогично, использование правила перестановки при подтиповировании компилируется в фрагмент кода, который физически изменяет порядок полей записи. После этого элементарные численные операции и доступ к полям могут выполняться без дополнительных расходов на снятие тегов или поиск.

С интуитивной точки зрения, семантика, основанная на преобразованиях типа, для языка с подтипами выражается в виде функции, которая переводит термы этого языка в язык нижнего уровня, в котором подтипы отсутствуют. В конечном счете, этот низкоуровневый язык может быть машинным кодом некоторого конкретного процессора. Однако в целях изложения мы продолжим обсуждение на более абстрактном уровне. В качестве исходного языка мы возьмём язык, используемый нами в большей части этой главы, — простое типизированное лямбда-исчисление с подтипами и записями. В качестве низкоуровневого целевого языка мы выбираем язык чистого простого типизированного лямбда-исчисления с записями и типом `Unit` (который используется для моделирования `Top`).

Формально компиляция представляется тремя функциями перевода — для типов, для подтиповирования и для типизации. В случае типов функция перевода просто заменяет `Top` типом `Unit`. Мы записываем эту функцию в виде $\llbracket - \rrbracket$.

$$\begin{aligned}\llbracket \text{Top} \rrbracket &= \text{Unit} \\ \llbracket T_1 \rightarrow T_2 \rrbracket &= \llbracket T_1 \rrbracket \rightarrow \llbracket T_2 \rrbracket \\ \llbracket \{l_i : T_i^{i \in 1..n}\} \rrbracket &= \{l_i : \llbracket T_i \rrbracket^{i \in 1..n}\}\end{aligned}$$

Например, $\llbracket \text{Top} \rightarrow \{a : \text{Top}, b : \text{Top}\} \rrbracket = \text{Unit} \rightarrow \{a : \text{Unit}, b : \text{Unit}\}$. (Остальные функции перевода также будут обозначаться символами $\llbracket - \rrbracket$; из контекста всегда будет ясно, какая именно функция имеется в виду.)

³ Аналогичные наблюдения применимы и к поиску полей и методов объектов в языках, в которых подтиповирование объектов допускает перестановку полей. Именно поэтому, например, Java ограничивает подтиповирование между классами так, что новые поля могут добавляться только в конце. Подтиповирование между интерфейсами (а также между классом и интерфейсом) разрешает перестановку — иначе интерфейсы оказались бы практически бесполезны. При этом руководство по языку явно предупреждает, что поиск метода в интерфейсе в общем случае происходит медленнее, чем в классе.

Чтобы перевести терм, мы должны знать, где при выводе его типа используется правило включения, поскольку именно в этих местах будет добавлено преобразование типа во время выполнения. Один из удобных способов формализации этого наблюдения состоит в определении перевода как функции на *деревьях вывода* утверждений о типизации. Аналогично, чтобы породить функцию преобразования, переводящую значения типа S в тип T , мы должны знать не только сам факт того, что S является подтиповом T , но и *почему* это так. Мы добиваемся этого, порождая преобразования из деревьев вывода подтиповирования.

Для формализации перевода требуется дополнительное соглашение по именованию деревьев вывода. Запись $\mathcal{C} :: S <: T$ будет означать: « \mathcal{C} — дерево вывода подтиповирования с заключением $S <: T$ ». Аналогично, $\mathcal{D} :: \Gamma \vdash t : T$ будет значить: « \mathcal{D} — дерево вывода типов с заключением $\Gamma \vdash t : T$ ».

Рассмотрим сначала функцию, которая на основе дерева вывода \mathcal{C} для утверждения о подтиповировании $S <: T$ порождает преобразование $\llbracket \mathcal{C} \rrbracket$. Это преобразование — не что иное как функция (в целевом языке перевода λ_{\rightarrow}) из типа $\llbracket S \rrbracket$ в тип $\llbracket T \rrbracket$. Определим его, рассматривая по отдельности каждый случай финального правила, используемого в \mathcal{C} .

$$\begin{aligned}
 \left[\frac{}{\overline{T <: T} \quad (\text{S-REFL})} \right] &= \lambda x: \llbracket T \rrbracket . \ x \\
 \left[\frac{}{\overline{S <: \text{Top}} \quad (\text{S-TOP})} \right] &= \lambda x: \llbracket S \rrbracket . \ \text{unit} \\
 \left[\frac{\mathcal{C}_1 :: S <: U \quad \mathcal{C}_2 :: U <: T}{S <: T} \quad (\text{S-TRANS}) \right] &= \lambda x: \llbracket S \rrbracket . \ [\mathcal{C}_2] (\llbracket \mathcal{C}_1 \rrbracket x) \\
 \left[\frac{\mathcal{C}_1 :: T_1 <: S_1 \quad \mathcal{C}_2 :: S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{S-ARROW}) \right] &= \lambda f: \llbracket S_1 \rightarrow S_2 \rrbracket . \ \lambda x: \llbracket T_1 \rrbracket . \ [\mathcal{C}_2] (\llbracket \mathcal{C}_1 \rrbracket x) \\
 \left[\frac{\{l_i: T_i^{i \in 1..n+k}\} <: \{l_i: T_i^{i \in 1..n}\}}{\{l_i: S_i^{i \in 1..n}\} <: \{l_i: T_i^{i \in 1..n}\}} \quad (\text{S-RCDWIDTH}) \right] &= \lambda r: \{l_i: \llbracket T_i \rrbracket^{i \in 1..n+k}\} . \ \{l_i = r . l_i^i\} \\
 \left[\frac{\text{для каждого } i \quad \mathcal{C}_i :: S_1 <: T_i}{\{l_i: S_i^{i \in 1..n}\} <: \{l_i: T_i^{i \in 1..n}\}} \quad (\text{S-RCDDEPTH}) \right] &= \lambda r: \{l_i: \llbracket S_i \rrbracket^{i \in 1..n}\} . \ \{l_i = \llbracket \mathcal{C}_i \rrbracket (r . l_i^i)\} \\
 \left[\frac{\{k_j: S_j^{j \in 1..n}\} \text{ перестановка } \{l_i: T_i^{i \in 1..n}\}}{\{k_j: S_j^{j \in 1..n}\} <: \{l_i: T_i^{i \in 1..n}\}} \quad (\text{S-RCDPERM}) \right] &= \lambda r: \{k_j: \llbracket S_j \rrbracket^{j \in 1..n}\} . \ \{l_i = r . l_i^i\}
 \end{aligned}$$

Лемма 15.6.1 Если $\mathcal{C} :: S <: T$, то $\vdash \llbracket \mathcal{C} \rrbracket : \llbracket S \rrbracket \rightarrow \llbracket T \rrbracket$.

Доказательство: Прямолинейная индукция по \mathcal{C} .

Деревья вывода типизации переводятся аналогично. Если \mathcal{D} — дерево вывода для утверждения $\Gamma \vdash t : T$, то его перевод $\llbracket \mathcal{D} \rrbracket$ — терм целевого языка типа $\llbracket T \rrbracket$. Такая функция перевода часто называется *Пенсильванской трансляцией* (Penn translation), в честь исследовательской группы Пенсильванского университета, которая впервые ее исследовала (Breazu-Tannen, Coquand, Gunter, and Scedrov, 1991).

$$\begin{aligned}
 & \left[\left[\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR}) \right] \right] = x \\
 & \left[\left[\frac{\mathcal{D}_2 :: \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-Abs}) \right] \right] = \lambda x: \llbracket T_1 \rrbracket. \llbracket \mathcal{D}_2 \rrbracket \\
 & \left[\left[\frac{\mathcal{D}_1 :: \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \mathcal{D}_2 :: \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1. t_2 : T_{12}} \quad (\text{T-APP}) \right] \right] = \llbracket \mathcal{D}_1 \rrbracket \llbracket \mathcal{D}_2 \rrbracket \\
 & \left[\left[\frac{\text{для каждого } i, \mathcal{D}_i :: \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i=t_i^{i \in 1..n}\} : \{l_i:T_i^{i \in 1..n}\}} \quad (\text{T-RCD}) \right] \right] = \{l_i=\llbracket \mathcal{D}_i \rrbracket^{i \in 1..n}\} \\
 & \left[\left[\frac{\mathcal{D}_1 :: \Gamma \vdash t_1 : \{l_i:T_i^{i \in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \quad (\text{T-PROJ}) \right] \right] = \llbracket \mathcal{D}_1 \rrbracket . l_j \\
 & \left[\left[\frac{\mathcal{D} :: \Gamma \vdash t : S \quad \mathcal{C} :: S <: T}{\Gamma \vdash t : T} \quad (\text{T-SUB}) \right] \right] = \llbracket \mathcal{C} \rrbracket \llbracket \mathcal{D} \rrbracket
 \end{aligned}$$

Теорема 15.6.2 *Если $\mathcal{D} :: \Gamma \vdash t : T$, то $\llbracket \Gamma \rrbracket \vdash \llbracket \mathcal{D} \rrbracket : \llbracket T \rrbracket$, где $\llbracket \Gamma \rrbracket$ — поточечное расширение функции перевода типов на контексты типизации: $\llbracket \emptyset \rrbracket = \emptyset$ и $\llbracket \Gamma, x:T \rrbracket = \llbracket \Gamma \rrbracket, x:\llbracket T \rrbracket$.*

Доказательство: Прямолинейная индукция по \mathcal{D} , с использованием леммы 15.6.1 для варианта T-SUB.

Определив эти функции перевода, можно отказаться от правил вычисления для высокоуровневого языка с подтиปами и вместо этого вычислять термы, подвергая их сначала проверке типов (через правила типизации и подтипирования для высокоуровневого языка), переводя деревья проверки типов в низкоуровневый целевой язык, и, наконец, определяя их операционное поведение через отношение вычисления для этого низкоуровневого языка. Такая стратегия действительно используется в некоторых высокопроизводительных реализациях языков с подтипами: например, в экспериментальном компиляторе языка Java, разработанном Йельской группой (League, Shao, and Trifonov, 1999; League, Trifonov, and Shao, 2001).

Упражнение 15.6.3 $[\star \star \star, \rightarrow]$: Модифицируйте приведенную нами функцию перевода так, чтобы целевым языком служило простое типизированное лямбда-исчисление с кортежами (а не с записями). Убедитесь, что теорема 15.6.2 по-прежнему выполняется.

Когерентность

При разработке семантики подтипов, основанной на преобразовании типов, необходимо избегать одной потенциальной ловушки. Например, предположим, что мы расширяем наш язык базовыми типами `Int`, `Bool`, `Float` и `String`. Нам могут пригодиться следующие элементарные преобразования:

$$\begin{aligned} \llbracket \text{Bool} <: \text{Int} \rrbracket &= \lambda b:\text{Bool}. \text{ if } b \text{ then } 1 \text{ else } 0 \\ \llbracket \text{Int} <: \text{String} \rrbracket &= \text{intToString} \\ \llbracket \text{Bool} <: \text{Float} \rrbracket &= \lambda b:\text{Bool}. \text{ if } b \text{ then } 1.0 \text{ else } 0.0 \\ \llbracket \text{Float} <: \text{String} \rrbracket &= \text{floatToString} \end{aligned}$$

Здесь `intToString` и `floatToString` — примитивы, генерирующие строковое представление чисел. Для примера предположим, что `intToString(1) = "1"`, а `floatToString(1.0) = "1.000"`.

Предположим теперь, что мы пытаемся вычислить терм

$(\lambda x:\text{String}.x) \text{ true}$

с помощью семантики преобразования типов. Этот терм типизируем, если принять во внимание вышеперечисленные аксиомы. Однако, его можно типизировать двумя разными способами: либо мы переводим `Bool` в `Int`, и затем в `String` через включение, показывая, что `true` — подходящий аргумент для функции типа `String → String`, либо сначала мы переводим `Bool` во `Float`, и оттуда в `String`. Однако при переводе этих преобразований в λ -предикат получается различное поведение. Если мы переводим `true` в `Int`, получается 1, откуда через `intToString` выходит строка "1". Однако если мы сначала проведем преобразование во `Float`, а затем, через `floatToString`, в `String` (следуя структуре дерева вывода типа, где `true : String` доказывается через `Float`), мы получим "1.000". Но "1" и "1.000" — совершенно различные строки; даже длина у них разная. Другими словами, выбор способа доказательства для $\vdash (\lambda x:\text{String}.x) \text{ true} : \text{String}$ влияет на то, как ведет себя оттранслированная программа! Однако этот выбор целиком содержится внутри компилятора — программист работает только с *термами*, а не с деревьями вывода, — что означает, что мы спроектировали язык, который не позволяет программисту не только управлять поведением написанных им программ, но даже и предсказывать его.

Правильное решение проблем такого рода заключается во введении *когерентности* (coherence) — дополнительного требования, которое налагается на определения функций перевода.

Определение 15.6.4 *Функция перевода $\llbracket - \rrbracket$ из деревьев вывода типов одного языка в термы другого когерентна, если для всякой пары деревьев вывода \mathcal{D}_1 и \mathcal{D}_2 с одним и тем же заключением $\Gamma \vdash t : T$ переводы $\llbracket \mathcal{D}_1 \rrbracket$ и $\llbracket \mathcal{D}_2 \rrbracket$ являются поведенчески эквивалентными термами целевого языка.*

В частности, вышеприведенные функции перевода (без базовых типов) когерентны. Чтобы восстановить когерентность после добавления базовых типов (с вышеуказанными аксиомами), достаточно изменить определение элементарной функции `floatToString` так, чтобы выполнялось `floatToString(0.0) = "0"` и `floatToString(1.0) = "1"`.

Доказательство когерентности, особенно для более сложных языков, может быть нелегкой задачей. См. работы Рейнольдса (Reynolds, 1980), Бреазу-Таннена и др. (Breazu-Tannen, Coquand, Gunter, and Scderv, 1991), Куриена и Гелли (Curien and Ghelli, 1992) и Рейнольдса (Reynolds, 1991).

15.7. Типы-пересечения и типы-объединения

Можно существенно уточнить отношение подтиповирования, если добавить в язык типов операцию *пересечения* (intersection). Типы-пересечения изобретены Коппо, Дезани, Салле и Поттингером (Coppo and Dezani-Ciancaglini, 1978; Coppo, Dezani-Ciancaglini, and Sallé, 1979; Pottinger, 1980). Доступное введение в эту тему можно найти у Рейнольдса (Reynolds, 1988, 1998b), Хиндли (Hindley, 1992) и Пирса (Pierce, 1991b).

Тип-пересечение $T_1 \wedge T_2$ содержит термы, принадлежащие к *обоим* типам T_1 и T_2 — то есть, $T_1 \wedge T_2$ является, в смысле теории порядков, пересечением (точной нижней гранью) T_1 и T_2 . Такая интуиция отражается в трех новых правилах подтиповирования:

$$T_1 \wedge T_2 \lessdot: T_1 \quad (\text{S-INTER1})$$

$$T_1 \wedge T_2 \lessdot: T_2 \quad (\text{S-INTER2})$$

$$\frac{S \lessdot: T_1 \quad S \lessdot: T_2}{S \lessdot: T_1 \wedge T_2} \quad (\text{S-INTER3})$$

Одно дополнительное правило отражает естественную взаимосвязь между типами-пересечениями и функциональными типами.

$$S \rightarrow T_1 \wedge S \rightarrow T_2 \lessdot: S \rightarrow (T_1 \wedge T_2) \quad (\text{S-INTER4})$$

Интуитивная идея, лежащая в основе этого правила, такова: если мы знаем, что терм имеет функциональные типы $S \rightarrow T_1$ и $S \rightarrow T_2$, то мы, безусловно, можем передать в эту функцию терм типа S и ожидать в качестве результата как T_1 , так и T_2 .

Мощность понятия типов-пересечений можно продемонстрировать утверждением, что в варианте простого типизированного лямбда-исчисления с подтипами, типами-пересечениями и вызовом по имени, множество нетипизированных лямбда-термов, которым можно присвоить типы, в точности совпадает с множеством *нормализуемых* (normalizing) термов. Т. е., терм типизируем тогда и только тогда, когда его вычисление завершается! Отсюда немедленно следует, что проблема реконструкции типов (см. главу 22) для исчисления с типами-перечислениями неразрешима.

С pragматической точки зрения ценность типов-пересечений состоит в том, что они поддерживают некоторую форму *конечной перегрузки операторов* (finitary overloading). Например, мы можем присвоить тип $(\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) \wedge (\text{Float} \rightarrow \text{Float} \rightarrow \text{Float})$ такой операции сложения, которую можно применить как к натуральным числам, так и к числам с плавающей

точкой (например, используя теговые биты в представлении аргументов для выбора правильной машинной команды во время выполнения).

К сожалению, мощность типов-пересечений ведет к некоторым практическим трудностям для разработчиков языков. Пока что лишь один полноценный язык, Forsythe (Reynolds, 1988), содержит типы-пересечения в их наиболее общем виде. Ограниченнная форма, называемая *типы-уточнения* (refinement types), может оказаться более разумной (Freeman and Pfenning, 1991; Pfenning, 1993b; Davies, 1997).

Двойственное понятие *типов-объединений* (union types) $T_1 \vee T_2$ также оказывается весьма полезным. В отличие от типов-сумм и вариантов (которые также иногда называют «объединениями», что приводит к путанице), $T_1 \vee T_2$ обозначает обычное объединение множества значений, принадлежащих к типу T_1 , и множества значений, принадлежащих к типу T_2 , безо всякого тега, указывающего на источник данного элемента. Таким образом, `Nat` \vee `Nat` — всего лишь псевдоним для `Nat`. Пересекающиеся типы-объединения уже давно играют важную роль при анализе программ (Palsberg and Pavlopoulou, 1998), но до сих пор редко встречались в языках программирования (важное исключение — Algol 68, см. van Wijngaarden, Mailloux, Peck, Koster, Sintzoff, Lindsey, Meertens, and Fisker, 1975). Впрочем, в последнее время их все чаще применяют в контексте систем типов для обработки «полуструктурированных» форматов баз данных, таких как XML (Buneman and Pierce, 1998; Hosoya, Vouillon, and Pierce, 2001).

Основное формальное различие между непересекающимися и пересекающимися типами-объединениями состоит в том, что для последних отсутствует какой-либо аналог конструкции `case`: если известно, что значение v имеет тип $T_1 \vee T_2$, то мы можем безопасно применять к нему только те операции, которые имеют смысл *как* для T_1 , *так и* для T_2 . (Например, если T_1 и T_2 — записи, то можно обращаться к их общим полям.) Бестеговый тип `union` в языке C приводит к нарушению типовой безопасности именно потому, что в языке игнорируется это ограничение, и над элементом типа $T_1 \vee T_2$ разрешена любая операция, имеющая смысл *либо* для T_1 , *либо* для T_2 .

15.8. Дополнительные замечания

Идея подтипов в языках программирования восходит к 1960-м годам, к языку Simula (Birtwistle, Dahl, Myhrhaug, and Nygaard, 1979) и его родственникам. Первые попытки формального анализа принадлежат Рейнольдсу (Reynolds, 1980) и Карделли (Cardelli, 1984).

Правила типизации и, особенно, подтиповования, работающие с записями, несколько сложнее, чем те, что мы видели до сих пор, поскольку в них либо имеется переменное число предпосылок (по одной на поле), либо используются дополнительные механизмы вроде перестановок индексов полей. Существует множество других способов записи этих правил, однако все они либо столь же сложны, либо избегают сложности, полагаясь на неформальные соглашения (например, многоточие: « $l_1:T_1 \dots l_n:T_n$ »). Недовольство таким положением дел привело Карделли и Митчелла к разработке исчисле-

ния *операций над записями* («Operations on Records», Cardelli and Mitchell, 1991), в котором макрооперация создания записи с многими полями выражается через базовое значение — пустую запись, а также операцию добавления одного поля за раз. С этой точки зрения можно также рассмотреть дополнительные операции, вроде изменения значения поля или конкатенации записей (Hargre and Pierce, 1991). Правила типизации для этих операций оказываются достаточно тонкими, особенно при наличии параметрического полиморфизма, так что разработчики языков предпочтитаю работать с обычновенными записями. Тем не менее, система Карделли и Митчелла остается важным концептуальным достижением. Альтернативный анализ записей, основанный на *полиморфизме строчных переменных* (row-variable polymorphism), разработан Вандом (Wand, 1987, 1988, 1989b), Реми (Rémy, 1990, 1989; Rémy, 1992) и другими, и служит основой для объектно-ориентированных конструкций языка OCaml (Rémy and Vouillon, 1998; Vouillon, 2000).

Основная задача, решаемая теорией типов, состоит в том, чтобы обеспечить наличие смысла у программ. Основная проблема, порождаемая теорией типов, состоит в том, что некоторым осмысленным программам никакого смысла не приписывается. Поиск более богатых систем типов проистекает из этого противоречия.

Марк Мэнасс

Глава 16

Метатеория подтипов

Определение простого типизированного лямбда-исчисления с подтипами из предыдущей главы не подходит для непосредственной реализации. В отличие от других рассмотренных нами исчислений, правила в этой системе не *управляются синтаксисом* (syntax directed) — их нельзя просто «прочитать снизу вверх», в результате получив алгоритм проверки типов. Сложности для отношения типизации происходят в основном из-за правила включения T-SUB, а для отношения подтипов — из-за правила транзитивности S-TRANS.

T-SUB вызывает трудности из-за того, что терм в его заключении выглядит как простая метапеременная t :

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad (\text{T-SUB})$$

Все остальные правила типизации содержат в заключении терм некоторого определенного вида: T-ABS относится только к лямбда-абстракциям, T-VAR только к переменным, и т. д., — а T-SUB применимо к термам *любого* вида. Это значит, что если у нас есть терм t и требуется вычислить его тип, то всегда будет можно применить как правило T-SUB, так и другое правило, заключение которого соответствует форме t .

Правило S-TRANS вызывает затруднения по той же самой причине — его заключение пересекается с заключениями других правил.

$$\frac{S <: U \quad U <: T}{S <: T} \quad (\text{S-TRANS})$$

Поскольку S и T — просто метапеременные, то в принципе, S-TRANS может оказаться последним правилом при выводе любого утверждения о подтиповании. Таким образом, наивная реализация правил подтипования «снизу

В этой главе изучается простое типизированное лямбда-исчисление с подтипами (рис. 15.1) и записями (рис. 15.3). Соответствующая реализация на OCaml называется `rsubst`. В §16.3 упоминаются также булевские значения и условные выражения (рис. 8.1); для этого раздела используется реализация `joinsub`. В §16.4 добавляется тип `Bot`; соответствующая реализация называется `bot`.

вверх» не сможет решить, следует ли попытаться применить это правило, либо же другое правило, более конкретное заключение которого также соответствует форме тех двух типов, членство которых в отношении подтиповирования мы пытаемся установить.

Трудности, связанные с S-TRANS, этим не исчерпываются. Обе его предпосылки содержат метапеременную U, которая не упоминается в заключении. Если мы наивно читаем правило снизу вверх, то получается, что мы должны *угадать* тип U, а затем попытаться доказать утверждения $S <: U$ и $U <: T$. Так как число типов-кандидатов на роль U бесконечно, надежда на успех при такой стратегии невелика.

Заключение правила S-REFL также пересекается с другими правилами подтиповирования. Этот случай проще, чем случай T-SUB и S-TRANS: в правиле S-REFL нет предпосылок, поэтому, если оно соответствует утверждению о подтиповировании, которое следует доказать, то мы заканчиваем немедленно. Однако, это еще одна причина, почему наши правила не управляются синтаксисом.

Чтобы справиться со всеми этими проблемами, заменим обычновенные (*декларативные*, declarative) отношения типизации и подтиповирования двумя новыми отношениями: *алгоритмическим отношением подтиповирования* (algorithmic subtyping) и *алгоритмическим отношением типизации* (algorithmic typing). Списки правил вывода этих двух отношений управляются синтаксисом. Чтобы обосновать эту замену, покажем затем, что исходные отношения подтиповирования и типизации совпадают с соответствующими алгоритмическими вариантами: утверждение $S <: T$ тогда и только тогда выводимо из алгоритмических правил подтиповирования, когда оно выводимо из декларативных правил, а терм тогда и только тогда типизируется по алгоритмическим правилам, когда он типизируется согласно декларативным правилам типизации.

Алгоритмическое отношение подтиповирования представлено в §16.1, а алгоритмическое отношение типизации — в §16.2. В §16.3 рассматривается особая проблема проверки ветвящихся конструкций вроде `if...then...else`, для которых требуется дополнительная структура (наличие точных верхних граней, *объединений* (joins), в отношении подтиповирования). В §16.4 рассматривается наименьший тип Bot.

16.1. Алгоритмическое отношение подтиповирования

При реализации любого языка с подтипами ключевую роль играет алгоритм для определения того, является ли один тип подтиром другого. Процедура проверки подтипов вызывается программой проверки типов, например, когда требуется обработать терм-применение $t_1 t_2$, где t_1 имеет тип $T \rightarrow U$, а t_2 — тип S . Задача процедуры проверки подтипов — определить, выводимо ли утверждение $S <: T$ из правил подтиповирования по рис. 15.1 и 15.3. Задача эта решается путем проверки принадлежности пары (S , T) другому отношению, которое обозначается $\mapsto S <: T$ (« S является подтиром T алгоритмической

$\rightarrow \{\} <:$

Расширяет 15.1 и 15.3

$S <: S$	(S-REFL)	$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$ (S-ARROW)
$\frac{S <: U \quad U <: T}{S <: T}$	(S-TRANS)	
$S <: \text{Top}$	(S-TOP)	$\frac{\{l_i : l_i^{i \in 1..n}\} \subseteq \{k_j : k_j^{j \in 1..m}\} \text{ при каждом } k_j = l_i \text{ имеем } S_j <: T_i}{\{k_j : S_j^{j \in 1..m}\} <: \{l_i : T_i^{i \in 1..n}\}}$ (S-RCD)

Рис. 16.1. Отношение потипирования для записей (компактная версия)

точки зрения). Это отношение определено так, что членство в нем можно определить, просто рассматривая структуру типов, и содержит те же самые пары типов, что и исходное отношение подтиповирования. Существенное различие между декларативным и алгоритмическим отношениями подтиповирования состоит в том, что в определении алгоритмического отношения нет правил S-TRANS и S-REFL.

Для начала требуется немного перестроить декларативную систему правил. Как мы видели на с. 208, транзитивность необходима, чтобы «склеивать» дерева вывода подтиповирования для записей, при одновременном использовании подтиповирования в ширину, в глубину и с перестановкой полей. Прежде чем мы откажемся от S-TRANS, нужно добавить правило, объединяющее эти три вида подтиповирования:

$$\frac{\{l_i : l_i^{i \in 1..n}\} \subseteq \{k_j : k_j^{j \in 1..m}\} \text{ при каждом } k_j = l_i \text{ имеем } S_j <: T_i}{\{k_j : S_j^{j \in 1..m}\} <: \{l_i : T_i^{i \in 1..n}\}} \quad (\text{S-RCD})$$

Лемма 16.1.1 *Если $S <: T$ выводится из правил подтиповирования, включая S-RCDDEPTH, S-RCDWIDTH и S-RCDPERM, но не включая S-RCD, то это утверждение также можно вывести с использованием S-RCD (но не S-RCDDEPTH, S-RCDWIDTH и S-RCDPERM), и наоборот.*

Доказательство: Прямолинейная индукция по деревьям вывода.

Благодаря лемме 16.1.1, мы можем заменить правила S-RCDDEPTH, S-RCDWIDTH и S-RCDPERM на S-RCD. Полученная система представлена на рис. 16.1.

Теперь мы можем показать, что в системе по рис. 16.1 правила рефлексивности и транзитивности несущественны.

$\rightarrow \{\} <:$

Алгоритмическое отношение подтипования

$\mapsto S <: \text{Top}$

$\mapsto S <: T$

(SA-TOP)

$$\frac{\{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\} \quad \text{если } k_j = l_i, \text{ то } \mapsto S_j <: T_i}{\mapsto \{k_j : S_j^{j \in 1..m}\} <: \{l_i : T_i^{i \in 1..n}\}}$$

(SA-RCD)

$$\frac{\mapsto T_1 <: S_1 \quad \mapsto S_2 <: T_2}{\mapsto S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

(SA-ARROW)

Рис. 16.2. Алгоритмическое отношение подтипования

Лемма 16.1.2

- Для каждого типа S можно вывести, что $S <: S$, без использования S-REFL.
- Если утверждение $S <: T$ выводимо, то его можно вывести без использования S-TRANS.

Доказательство: Упражнение [Рекомендуется, ★★★].

Упражнение 16.1.3 [★] Как изменяются эти свойства при добавлении типа Bool?

Это приводит нас к определению алгоритмического отношения подтипования:

Определение 16.1.4 Отношением алгоритмического подтипования называется минимальное отношение между типами, замкнутое относительно правил по рис. 16.2.

Мы говорим, что алгоритмические правила *корректны* (sound), поскольку всякое утверждение, выводимое через них, можно вывести и при помощи декларативных правил (алгоритмические правила не доказывают ничего нового), а также *полны* (complete), поскольку каждое утверждение, выводимое из декларативных правил, может быть выведено и при помощи алгоритмических (алгоритмические правила способны доказать все, что было доказуемо раньше).

Утверждение 16.1.5 [КОРРЕКТНОСТЬ И ПОЛНОТА] $S <: T$ тогда и только тогда, когда $\mapsto S <: T$.

Доказательство: В обоих направлениях доказательство строится по индукции на деревьях вывода, с помощью одной из двух предыдущих лемм.

Поскольку алгоритмические правила управляются синтаксисом, их можно непосредственно рассматривать как алгоритм проверки отношения алгоритмического подтиповирования (а следовательно, и декларативного отношения подтиповирования). В псевдокоде алгоритм выглядит так:

```

subtype (S, T) = 
    если T = Тор, то истина
    иначе если S = S1 → S2 и T = T1 → T2
        то subtype (T1, S1) ∧ subtype (S2, T2)
    иначе если S = {kj: Sjj ∈ 1..m} и T = {li: Tii ∈ 1..n}
        то {lii ∈ 1..n} ⊆ {kjj ∈ 1..m}
            ∧ для каждого i имеется некоторое j ∈ 1..m такое, что kj = li
            и subtype (Sj, Ti)
    иначе ложь
```

Конкретную реализацию этого алгоритма на языке ML можно найти в главе 17.

Наконец, нам нужно убедиться в том, что алгоритмическое отношение подтиповирования *всюду определено* (total), т. е. что рекурсивная функция *subtype*, построенная на основе алгоритмических правил, возвращает для каждой пары входных параметров либо значение *истина*, либо *ложь*, причем за конечное время.

Утверждение 16.1.6 [ЗАВЕРШЕНИЕ] *Если выводимо утверждение $\mapsto S <: T$, то функция subtype(S, T) вернет значение истина. В противном случае она вернет значение ложь.*

Эта теорема, совместно с утверждениями о корректности и полноте алгоритмических правил, по существу, утверждает, что функция *subtype* служит разрешающей процедурой (*decision procedure*) для декларативного отношения подтиповирования.

Доказательство: Первое утверждение легко поддается проверке (путем прямолинейной индукции по дереву вывода $\mapsto S <: T$). С другой стороны, так же легко убедиться, что, если *subtype*(S, T) возвращает значение истина, то выполняется $\mapsto S <: T$. Таким образом, чтобы проверить второе утверждение, достаточно показать, что *subtype*(S, T) всегда хоть что-то возвращает, т. е. всегда завершается. Этого мы можем добиться, заметив, что сумма размеров входной пары аргументов S и T всегда строго больше, чем сумма размеров аргументов в любом рекурсивном вызове, производимом алгоритмом. Поскольку эта сумма — всегда конечное положительное число, бесконечная последовательность рекурсивных вызовов невозможна.

Читатель спросит, нельзя ли сократить работу, проделанную нами в этом разделе, если сразу взять за основу алгоритмическое определение отношения подтиповирования, и даже не упоминать декларативное определение. Ответом будет «можно, но с оговорками». Разумеется, при желании мы можем выбрать для определения нашего исчисления алгоритмическое отношение в качестве основного. Однако в результате мы почти не экономим усилия, поскольку, чтобы показать, что отношение типизации (зависящее от отношения подтиповирования) ведет себя корректно, необходимо знать, что отношение подтипи-

рования рефлексивно и транзитивно, а эти доказательства требуют примерно такого же объема работы, какой мы проделали здесь. (С другой стороны, в определениях языков часто используется алгоритмическое представление отношения типизации. Пример этого мы увидим в главе 19.)

16.2. Алгоритмическое отношение типизации

Разобравшись с отношением подтипов, мы должны теперь проделать то же самое с отношением типизации. Как мы видели на с. 235, единственное правило типизации, не управляемое синтаксисом, — это T-SUB, так что именно с ним нам и предстоит иметь дело. Как и в случае S-TRANS в предыдущем разделе, мы не можем просто убрать из системы правило включения: нужно сначала выяснить, где это правило играет существенную роль, и расширить другие правила так, чтобы те же самые результаты достигались более синтаксически управляемым способом.

Очевидно, что один важный случай, в котором применяется правило включения, — это установление связей между типами аргументов, которые ожидает функция, и типами её реальных аргументов. Такой терм, как

$$(\lambda r : \{x : \text{Nat}\}. \ r.x) \ \{x=0, y=1\}$$

без правила включения не типизируется.

Возможно, это покажется странным, но указанная ситуация — *единственная*, где правило включения необходимо при типизации. Во всех остальных случаях, когда при доказательстве типизации встречается это правило, то же самое утверждение можно получить из другого дерева вывода, в котором использование T-SUB «откладывается», будучи сдвинуто вниз по дереву к корню. Чтобы в этом убедиться, полезно немного поэкспериментировать с деревьями вывода, содержащими правило включения, рассматривая каждое правило типизации по очереди и размышляя, как можно перестроить вывод, завершающийся этим правилом, так чтобы один из непосредственных его подвыводов завершался бы применением T-SUB.

Предположим, например, что у нас есть дерево вывода, завершающееся правилом T-ABS, причем его непосредственный подвывод заканчивается применением T-SUB:

$$\frac{\begin{array}{c} \vdots \\ \Gamma, x:S_1 \vdash s_2 : S_2 \quad S_2 <: T_2 \end{array}}{\Gamma, x:S_1 \vdash s_2 : T_2} \text{ (T-SUB)} \quad \frac{\Gamma, x:S_1 \vdash s_2 : T_2}{\Gamma \vdash \lambda x:S_1.s_2 : S_1 \rightarrow T_2} \text{ (T-ABS)}$$

Такое дерево можно перестроить так, чтобы правило включения использова-

лось *после* правила абстракции, и получить то же заключение, что и раньше:

$$\frac{\vdots}{\Gamma, x:S_1 \vdash s_2 : S_2} \text{ (T-Abs)} \quad \frac{\vdots}{S_1 <: S_1} \text{ (S-REFL)} \quad \frac{\vdots}{S_2 <: T_2} \text{ (S-ARROW)} \\ \frac{\Gamma \vdash \lambda x:S_1.s_2 : S_1 \rightarrow S_2 \quad S_1 \rightarrow S_2 <: S_1 \rightarrow T_2}{\Gamma \vdash \lambda x:S_1.s_2 : S_1 \rightarrow T_2} \text{ (T-S)}$$

Более интересный случай — правило применения Т-APP. Здесь имеется два подвывода, каждый из которых может заканчиваться правилом Т-SUB. Сначала рассмотрим случай, когда правило включения встречается в конце левого подвывода.

$$\frac{\vdots \quad \vdots}{\Gamma \vdash s_1 : S_{11} \rightarrow S_{12}} \quad \frac{S_{11} \rightarrow S_{12} <: T_{11} \rightarrow T_{12}}{\Gamma \vdash s_1 : T_{11} \rightarrow T_{12}} \text{ (T-SUB)} \\ \frac{\vdots}{\Gamma \vdash s_2 : T_{11}} \text{ (T-S)}$$

Учитывая результаты из предыдущего раздела, мы имеем право предположить, что последним правилом в выводе $S_{11} \rightarrow S_{12} <: T_{11} \rightarrow T_{12}$ не будет ни S-REFL, ни S-TRANS. Исходя из формы его заключения, таким правилом может быть только S-ARROW.

$$\frac{\vdots \quad \vdots}{\frac{T_{11} <: S_{11} \quad S_{12} <: T_{12}}{S_{11} \rightarrow S_{12} <: T_{11} \rightarrow T_{12}}} \text{ (S-ARROW)} \\ \frac{\vdots}{\Gamma \vdash s_1 : T_{11} \rightarrow T_{12}} \text{ (T-SUB)} \\ \frac{\vdots}{\Gamma \vdash s_1 s_2 : T_{12}} \text{ (T-S)}$$

Переписав правила и устранив Т-SUB, мы получаем интересный результат.

$$\frac{\vdots \quad \vdots}{\frac{\Gamma \vdash s_2 : T_{11} \quad T_{11} <: S_{11}}{\Gamma \vdash s_2 : S_{11}}} \text{ (T-SUB)} \\ \frac{\vdots}{\Gamma \vdash s_1 s_2 : S_{12}} \text{ (T-APP)} \\ \frac{\vdots}{\Gamma \vdash s_1 s_2 : T_{12}} \text{ (T-S)}$$

Правый подвывод исходного экземпляра S-ARROW сдвинут вниз в корень дерева, где новый экземпляр Т-SUB повышает тип всего терма-применения. С другой стороны, левый подвывод передвинут *вверх* по дереву, туда, где определяется тип аргумента s_2 .

Предположим теперь, что экземпляр T-SUB, подлежащий переносу, расположена в конце правого подвыхода дерева, завершающегося правилом T-APP.

$$\frac{\vdots \quad \vdots}{\frac{\Gamma \vdash s_2 : T_2 \quad T_2 <: T_{11}}{\Gamma \vdash s_2 : T_{11}}} \text{ (T-SUB)} \\ \frac{\Gamma \vdash s_1 : T_{11} \rightarrow T_{12}}{\Gamma \vdash s_1 \ s_2 : T_{12}} \text{ (T-APP)}$$

Единственное, что мы можем сделать с таким экземпляром T-SUB — это передвинуть его в левый подвыход, частично отменяя предыдущее преобразование.

$$\frac{\vdots \quad \vdots \quad \vdots}{\frac{\frac{T_2 <: T_{11}}{T_{12} <: T_{12}} \text{ (S-REFL)}}{\frac{\Gamma \vdash s_1 : T_{11} \rightarrow T_{12}}{\Gamma \vdash s_1 : T_2 \rightarrow T_{12}} \text{ (T-SUB)}} \quad \frac{\vdots}{\Gamma \vdash s_2 : T_2} \text{ (T-APP)}}{\Gamma \vdash s_1 \ s_2 : T_{12}}$$

Итак, как мы видим, экземпляр правила включения, который повышает тип результата терма-применения, может быть передвинут вниз сквозь правило T-APP. Однако, экземпляра, в котором включение используется для соглашения между типом аргумента функции и типом области ее определения, уничтожить нельзя. Можно передвинуть его из одной предпосылки в другую: так, мы можем расширить тип аргумента, чтобы он соответствовал типу области определения, либо расширить тип функции (сузив тип ее аргумента) так, чтобы она ожидала аргумент, который мы на самом деле собираемся в нее ввести. Однако, вообще избавиться от правила включения здесь невозможно. Это наблюдение в точности соответствует нашему интуитивному представлению о том, что такое использование T-SUB и делает нашу систему столь мощной.

Также требует рассмотрения случай, в котором последним правилом в дереве вывода является включение, а его непосредственный подвыход также завершается правилом включения. В этом случае два следующих подряд экземпляра включения можно слить в один. Т. е., всякий вывод вида

$$\frac{\vdots \quad \vdots}{\frac{\frac{\Gamma \vdash s : S \quad S <: U}{\Gamma \vdash s : U} \text{ (T-SUB)}}{\frac{\vdots}{\frac{U <: T}{\Gamma \vdash s : T}} \text{ (T-SUB)}}}$$

можно переписать как

$$\frac{\frac{\vdots \quad \vdots}{\frac{\vdots \quad \vdots}{\frac{\Gamma \vdash s : S}{\Gamma \vdash s : T}} \quad \frac{S <: U \quad U <: T}{S <: T} \text{ (S-TRANS)}} \text{ (T-SUB)}}$$

Упражнение 16.2.1 [★ →]: Для завершения эксперимента покажите, как производить подобное переупорядочение правил в выводах, в которых правило T-SUB используется перед T-RCD либо перед T-PROJ.

Применив эти преобразования шаг за шагом, мы можем привести произвольное дерево вывода типа к форме особого вида, в которой T-SUB встречается только в двух местах: в конце правого подвывода для термов-применений и в самом конце всего дерева. Более того, если мы просто *отбросим* экземпляр T-SUB в самом конце, никакого особенного вреда это не принесет: у нас по-прежнему останется дерево вывода, присваивающее тип тому же самому терму — с тем единственным отличием, что тип, присваиваемый терму, будет меньше (то есть лучше!). Остается лишь одно место — термы-применения, где все еще могут встречаться экземпляры правила включения. Чтобы учесть эти случаи, заменим правило для применений несколько более мощной версией,

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_2 \quad T_2 <: T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$$

уже содержащей использование включения в качестве предпосылки. Каждый подвывод вида «применение с включением перед ним» можно заменить на экземпляр этого правила, в результате чего у нас не останется никаких случаев использования T-SUB. Более того, расширенное правило применения управляетя синтаксисом: форма терма в его заключении не пересекается с остальными правилами.

Такое преобразование дает нам набор правил типизации, управляемый синтаксисом, и при этом присваивающий типы тем же самым термам, что и исходные правила типизации. Новые правила сведены в нижеследующем определении. Как и в случае с алгоритмическими правилами подтиповирования, мы записываем алгоритмическое отношение типизации с помощью символа «штопора», $\Gamma \vdash t : T$, чтобы не путать его с декларативным отношением.

Определение 16.2.2 Отношение алгоритмической типизации (*algorithmic typing relation*) есть наименьшее отношение, замкнутое относительно правил с рис. 16.3. Предпосылка $T_1 = T_{11} \rightarrow T_{12}$ в правилах для применений — просто явное напоминание о последовательности операций при проверке типов: мы сначала вычисляем тип T_1 для t_1 ; затем проверяем, что T_1 имеет вид $T_{11} \rightarrow T_{12}$, и т. д. Правило работало бы точно в тех же случаях, если бы мы исключили эту предпосылку, а вместо первой предпосылки написали бы $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$. То же замечание верно и для TA-PROJ. Кроме того,

$\rightarrow \{ \} <:$

Алгоритмическая типизация	(TA-VAR)	(TA-RCD)
$\boxed{\Gamma \vdash t : T}$	$\frac{x:T \in \Gamma}{\Gamma \vdash x : T}$	$\frac{\text{для каждого } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_1=t_1 \dots l_n=t_n\} : \{l_1:T_1 \dots l_n:T_n\}}$ (TA-RCD)
$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2}$	(TA-ABS)	$\frac{\Gamma \vdash t_1 : R_1 \quad R_1 = \{l_1:T_1 \dots l_n:T_n\}}{\Gamma \vdash t_1.l_i : T_i}$ (TA-PROJ)
$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_2 \quad \vdash T_2 <: T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}}$	(TA-APP)	

Рис. 16.3. Алгоритмическая типизация

предпосылка о подтипности в правиле для применений записана с использованием символа «штопора»; поскольку мы знаем, что алгоритмическое отношение подтипирования эквивалентно декларативному, такой выбор — дело вкуса.

Упражнение 16.2.3 [★ →]: Покажите, что тип, присваиваемый терму алгоритмическими правилами, может уменьшиться при вычислении. Для этого найдите два терма s и t с алгоритмическими типами S и T , чтобы $s \rightarrow^* t$ и $T <: S$, но $S \not<: T$.

Нам еще нужно формально проверить, что алгоритмические правила типизации соответствуют исходным декларативным правилам. (Преобразования деревьев вывода, перечисленные нами выше, слишком неформальны, чтобы считаться доказательством. Их можно превратить в такое доказательство, но такая работа будет слишком долгой и трудоемкой: проще, как обычно, построить индукцию по деревьям вывода.) Как и в случае с подтипированием, мы доказываем, что алгоритмические правила корректны и полны по отношению к исходным декларативным правилам.

Свойство корректности имеет прежний вид: каждое утверждение о типизации, выводимое из алгоритмических правил, также следует и из правил декларативных.

Теорема 16.2.4 [КОРРЕКТНОСТЬ] Если $\Gamma \vdash t : T$, то $\Gamma \vdash t : T$

Доказательство: Прямолинейная индукция по алгоритмическим деревьям вывода типов.

Что касается свойства полноты, то оно выглядит немного иначе. Обыкновенное отношение типизации может присвоить терму множество типов, тогда как алгоритмическое присваивает не более одного (это нетрудно проверить). Таким образом, прямое преобразование теоремы 16.1.5 было бы явно неверным. Вместо этого мы можем показать, что если терм t имеет по обычным правилам тип T , то по алгоритмическим правилам у него всегда есть тип S , лучший, чем T в том смысле, что $S <: T$. Другими словами, алгоритмические правила присваивают каждому терму его *наименьший* (minimal) тип. Теорему о полноте часто называют теоремой о наименьшем типе, поскольку (вместе с теоремой 16.2.4) она означает, что всякий типизируемый терм в декларативной системе имеет наименьший тип.

Теорема 16.2.5 [Полнота, или Минимальность типов] *Если $\Gamma \vdash t : T$, то $\Gamma \vdash t : S$ для некоторого $S <: T$.*

Доказательство: Упражнение [Рекомендуется, ★★].

Упражнение 16.2.6 [★★]: *Если отбросить правило подтипования для функций S-ARROW, а остальные декларативные правила подтипования и типизации оставить неизменными, будет ли система по-прежнему обладать свойством минимальности типов? Если да, то докажите это. Если нет, то приведите пример типизируемого терма, у которого отсутствует наименьший тип.*

16.3. Пересечения и объединения

Проверка типов выражений с несколькими возможными результатами, таких как условные выражения или выражения `case`, в языке с подтипами требует некоторых дополнительных хитростей. Вспомним, например, декларативное правило типизации для условных выражений:

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$$

Правило требует, чтобы тип обеих ветвей был одинаков, и присваивает этот тип условному выражению в целом. Однако если в системе имеется правило включения, то может быть *много* способов дать обеим ветвям один и тот же тип. Например,

```
if true then {x=true,y=false} else {x=false,z=true}
```

имеет тип `{x:Bool}`, поскольку ветвь `then` имеет наименьший тип `{x:Bool,y:Bool}`, который можно поднять до `{x:Bool}` с помощью T-SUB, а ветвь `else`, аналогично, имеет наименьший тип `{x:Bool,z:Bool}`, который тоже можно поднять до `{x:Bool}`. Тот же самый терм имеет и типы `{x:Top}` и `{}` — в сущности, *любой* тип, одновременно являющийся надтипов `{x:Bool,y:Bool}` и `{x:Bool,z:Bool}`. Следовательно, наименьшим типом всего условного выражения является наименьший общий надтип `{x:Bool,y:Bool}` и `{x:Bool,z:Bool}`, т. е. `{x:Bool}`. В общем случае, чтобы вычислить наименьший тип произвольного условного выражения, следует

найти наименьшие типы его ветвей `then` и `else`, а затем найти их наименьший общий надтип. Такой тип часто называют *объединением* типов ветвей, поскольку он соответствует обычному объединению (точной верхней грани) двух элементов в частичном порядке.

Определение 16.3.1 *Тип J называется объединением (join) пары типов S и T , что записывается в виде $S \vee T = J$, если $S <: J$, $T <: J$, и для всех типов U , если $S <: U$ и $T <: U$, то $J <: U$. Аналогично, тип M называется пересечением (meet) S и T , что обозначается как $S \wedge T = M$, если $M <: S$, $M <: T$, и для всякого типа L , если $L <: S$ и $L <: T$, то $L <: M$.*

В зависимости от того, как в конкретном языке определено отношение подтиповирования, не всегда для всякой пары типов может существовать объединение. Говорят, что отношение подтиповирования *обладает объединениями*, если для всяких S и T существует некоторый тип J , являющийся объединением S и T . Аналогично, отношение подтиповирования *обладает пересечениями*, если для каждого S и T имеется некоторый M — пересечение S и T .

Отношение подтиповирования, рассматриваемое нами в этом разделе,¹ обладает объединениями, но не пересечениями. Например, у типов `{}` и `Top`—`Top` вообще нет общих подтипов, так что нет и наибольшего общего подтипа. Однако, несколько более слабое условие все ещё выполняется. Мы говорим, что пара типов S и T *ограничена снизу* (bounded below), если имеется тип L такой, что $L <: S$ и $L <: T$. Отношение подтиповирования *обладает ограниченными пересечениями* (has bounded meets), если для каждой пары ограниченных снизу типов S и T имеется M — пересечение S и T .

Объединения и пересечения не обязательно должны быть уникальны. Например, как `{x:Top, y:Top}`, так и `{y:Top, x:Top}` являются объединениями пары типов `{x:Top, y:Top, z:Top}` и `{x:Top, y:Top, w:Top}`. Однако два различных объединения (или пересечения) одной и той же пары типов всегда должны быть подтипами друг друга.

Утверждение 16.3.2 [Существование объединений и ограниченных пересечений]

1. Для каждой пары типов S и T имеется тип J , такой, что $S \vee T = J$.
2. Для каждой пары типов S и T , имеющих общий подтип, имеется тип M , такой, что $S \wedge T = M$.

Доказательство: УПРАЖНЕНИЕ [Рекомендуется, ★★★].

С помощью операции объединения мы можем задать алгоритмическое правило для выражений `if` в системе с подтипами:

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = \text{Bool} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3 \quad T_2 \vee T_3 = T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{TA-IF})$$

¹ А именно, отношение, определенное на рис. 15.1 и 15.3, и расширенное типом `Bool`. В отношении подтиповирования `Bool` ведет себя просто: в декларативное отношение подтиповирования никаких связанных с ним правил не добавляется, так что его единственным надтипом является `Top`.

Упражнение 16.3.3 [★★]: Каков наименьший тип выражения `if true then false else {}?` Хорошо ли это?

Упражнение 16.3.4 [★★★]: Легко ли расширить алгоритмы для вычисления типов-объединений и типов-пересечений на императивный язык со ссылками, описанными в §15.5 (на с. 222)? А если принять подход к ссылкам из §15.5 (с. 224), где мы разделяем инвариантный конструктор типов `Ref` на ковариантный `Source` и контравариантный `Sink`?

16.4. Алгоритмическая типизация и тип Bot

Если в отношение подтиповирования добавляется наименьший тип `Bot` (§15.4), требуется несколько расширить алгоритмы для подтиповирования и типизации. Добавим одно (очевидное) правило в отношение подтиповирования:

$$\vdash \text{Bot} <: T \quad (\text{SA-Bot})$$

и два несколько более сложных правила в алгоритмическое отношение типизации:

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = \text{Bot} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \ t_2 : \text{Bot}} \quad (\text{TA-APPBot})$$

$$\frac{\Gamma \vdash t_1 : R_1 \quad R_1 = \text{Bot}}{\Gamma \vdash t_1.l_i : \text{Bot}} \quad (\text{TA-ProjBot})$$

Правило для подтиповирования очевидно. Интуитивная идея, стоящая за правилами типизации, такова: в декларативной системе мы можем применить объект типа `Bot` к чему угодно (при помощи включения продвинув `Bot` до любого интересующего нас функционального типа), и предположить, что полученный результат имеет любой другой тип. Аналогично можно рассуждать и в случае проекции.

Упражнение 16.4.1 [★]: Допустим, что в языке также имеются условные выражения. Нужно ли добавить еще одно алгоритмическое правило для `if`?

Дополнения, требуемые для поддержки `Bot` в этом языке, не слишком сложны. Однако в §28.8 мы увидим, что при сочетании `Bot` с ограниченной квантификацией возникают более серьезные сложности.

Глава 17

Реализация подтипов на ML

В этой главе интерпретатор для простого типизированного лямбда-исчисления, разработанный в главе 10, расширяется дополнительными механизмами, необходимыми для поддержки подтипов — в частности, функцией для проверки отношения подтипования.

17.1. Синтаксис

Определения типов данных для типов и термов записаны с помощью абстрактного синтаксиса по рис. 15.1 и 15.3.

```
type ty =
  TyRecord of (string * ty) list
  | TyTop
  | TyArr of ty * ty

type term =
  | TmRecord of info * (string * term) list
  | TmProj of info * term * string
  | TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
```

По сравнению с чистым простым типизированным лямбда-исчислением, здесь добавлены тип `TyTop`, конструктор типов `TyRecord` и конструкторы термов `TmRecord` и `TmProj`. Записи и их типы представляются простейшим способом, в виде списка полей и связанных с ними термов либо типов.

17.2. Подтипы

Представление алгоритмического отношения подтипования в виде псевдокода со с. 239 прямо переводится на OCaml следующим образом:

```
let rec subtype tyS tyT =
  (=) tyS tyT ||
```

```

match (tyS, tyT) with
  (TyRecord(fS), TyRecord(fT)) →
    List.for_all
      (fun (li, tyTi) →
        try let tySi = List.assoc li fS in
          subtype tySi tyTi
        with Not_found → false)
      fT
  | (_, TyTop) →
    true
  | (TyArr(tyS1, tyS2), TyArr(tyT1, tyT2)) →
    (subtype tyT1 tyS1) && (subtype tyS2 tyT2)
  | (_, _) →
    false

```

По сравнению с псевдокодом мы внесли в алгоритм небольшое изменение, добавив в начало проверку на рефлексивность. (Операция `(=)` — это обыкновенная проверка на равенство; здесь она записана в префиксной нотации потому, что в некоторых других реализациях подтипов равенство проверяется вызовом другой функции. Операция `||` — это булевское «или» с оптимизацией: если первая ветвь дает значение «истина», то вторая не вычисляется). Строго говоря, эта проверка не нужна. Однако в реальных компиляторах эта оптимизация очень важна. В большинстве реальных программ подтипы используются довольно редко — в большинстве случаев, когда вызывается проверка на подтиповирование, сравниваемые типы на самом деле одинаковы. Более того, если типы представляются так, чтобы структурно одинаковые типы всегда имели физически идентичные представления — скажем, через *cons* с хэшированием (hash consing) (Goto, 1974; Appel and Gonçalves, 1993), — то эта проверка производится одной машинной командой.

Естественно, проверка на подтиповирование для записей требует некоторой работы со списками. Функция `List.for_all` применяет предикат (свой первый аргумент) к каждому члену списка, и возвращает `true`, если все эти применения дают результат `true`. Выражение `List.assoc li fS` ищет метку `li` в списке полей `fS` и возвращает соответствующий тип `tySi`; если же `li` не встречается среди меток `fS`, возникает исключение `Not_found`, которое мы перехватываем и превращаем в возвращаемое значение `false`.

17.3. Типизация

Функция проверки типов есть прямолинейное расширение функции `typeof` из предыдущих интерпретаторов. Основное изменение коснулось ветки с применением, поскольку требуется провести проверку на подтиповирование между типом аргумента и типом, который ожидает функция. Кроме того, мы добавляем две новых ветви для конструирования записей и их проекции.

```

let rec typeof ctx t =
  match t with
    TmRecord(fi, fields) →
      let fieldtys =

```

```

        List.map (fun (li,ti) → (li, typeof ctx ti)) fields in
        TyRecord(fieldtys)
| TmProj(fi, ti, l) →
    (match (typeof ctx ti) with
     TyRecord(fieldtys) →
        (try List.assoc l fieldtys
         with Not_found → error fi ("label^l^not_found"))
        | _ → error fi "Expected_record_type")
| TmVar(fi,i,_) → getTypeFromContext fi ctx i
| TmAbs(fi,x,tyT1,t2) →
    let ctx' = addbinding ctx x (VarBind(tyT1)) in
    let tyT2 = typeof ctx' t2 in
    TyArr(tyT1, tyT2)
| TmApp(fi,t1,t2) →
    let tyT1 = typeof ctx t1 in
    let tyT2 = typeof ctx t2 in
    (match tyT1 with
     TyArr(tyT11,tyT12) →
        if subtype tyT2 tyT11 then tyT12
        else error fi "parameter_type_mismatch"
     | _ → error fi "arrow_type_expected")

```

В ветвях, относящихся к записям, вводятся новые конструкции OCaml, которые прежде нам не встречались. В ветви `TmRecord` мы вычисляем список имен полей с их типами `fieldtys` из списка полей с термами `fields`, с помощью `List.map` применяя функцию

```
fun (li,ti) → (li, typeof ctx ti)
```

по очереди к каждой паре имя/терм. В ветви `TmProj` мы снова используем `List.assoc`, на этот раз чтобы найти тип выбранного поля. Если возникает исключение `Not_found`, мы порождаем наше собственное сообщение об ошибке (символ `^` обозначает конкатенацию строк).

Упражнение 17.3.1 [★★★]: В §16.3 показано, как при добавлении условных выражений в язык с подтипами возникает нужда в дополнительных вспомогательных функциях для вычисления наименьшей верхней грани данной пары типов. В доказательстве утверждения 16.3.2 (см. с. 553) дается математическое описание требуемых алгоритмов.

Программа проверки типов `joinexercise` — это неполная реализация простого типизированного лямбда-исчисления с подтипами, записями и условными выражениями: даны основные функции для синтаксического анализа и распечатки, но в функции `typeof` отсутствует вариант для `TmIf`. Нет также функции `join`, от которой этот вариант зависит. Добавьте в этот интерпретатор булевские значения и условные выражения (а также обединения и пересечения).

Упражнение 17.3.2 [★★]: Добавьте в интерпретатор `rcdsu` минимальный тип `Bot`, основываясь на описании, приведенном в §16.4.

Упражнение 17.3.3 [★ ★ ★, →]: Если проверка на подтиповование в правиле применения завершается неудачей, то печатаемое нашим интерпретатором сообщение об ошибке может быть непонятно пользователю. Можно его улучшить, распечатывая в составе сообщения об ошибке ожидаемый и реально имеющийся типы параметра, но даже такую запись может быть трудно понять. Например, если ожидается тип

```
{x:{}, y:{}, z:{}, a:{}, b:{}, c:{}, d:{}, e:{}, f:{}, g:{}}
```

а реально получен тип

```
{y:{}, z:{}, f:{}, a:{}, x:{}, i:{}, b:{}, e:{}, g:{}, c:{}, h:{}}
```

далеко не сразу можно сообразить, что пропущено поле *d*. Можно значительно улучшить диагностические сообщения, если заставить функцию *subtype* вместо значений *true* и *false* либо возвращать трибунальное значение (значение () типа *unit*), либо порождать исключение. Поскольку исключение порождается в том месте проверки типов, где реально обнаружено несоответствие, то мы можем точнее описать сущность ошибочной ситуации. Заметим, что это изменение не затрагивает наблюдаемое извне поведение программы проверки: если проверка поттипирования возвращает *false*, то процедура проверки типов все равно породила бы в этом месте другое исключение (через вызов функции *error*).

Перепишите функции *typeof* и *subtype* так, чтобы все сообщения об ошибках были как можно более информативны.

Упражнение 17.3.4 [★ ★ ★, →]: В §15.6 мы определили семантику на основе преобразования типов (*coercion semantics*) для языка с записями и подтипами, переводя выводы типизации и подтиповования в термы чистого простого типизированного лямбда-исчисления. Реализуйте эти преобразования, модифицировав функцию *subtype* из этой главы так, чтобы она порождала и возвращала функцию преобразования (выраженную в виде терма), а также изменив функцию *typeof* так, чтобы она возвращала как тип, так и переведенный терм. После этого нужно выполнить переведенный терм (а не исходный терм), а результат, как обычно, нужно распечатать.

Глава 18

Расширенный пример: императивные объекты

В этой главе мы построим первый пример нетривиальной системы. Мы используем почти все конструкции, определенные ранее — функции, записи, рекурсию общего вида, изменяемые ссылки и подтипы, — и с их помощью создадим набор программных идиом для поддержки объектов и классов вроде тех, что есть в объектно-ориентированных языках программирования, таких как Smalltalk или Java. В этой главе мы не добавляем новый синтаксис для объектов: нашей целью будет *понять* эти достаточно сложные языковые конструкции, показав, как их поведение можно реализовать на основе более низкоуровневых элементов языка.

На протяжении большей части главы реализация будет довольно точной: мы сможем получить удовлетворительную функциональность большинства возможностей объектов и классов, рассматривая их как производные формы, переводимые через удаление сахара в простые сочетания уже рассмотренных нами конструкций. Однако когда в §18.9 мы доберемся до виртуальных методов и `self`, мы столкнемся с некоторыми сложностями, связанными с порядком вычислений, и из-за этого подход с удалением сахара окажется несколько нереалистичным. Более удовлетворительное описание этих конструкций можно получить путем прямой аксиоматизации их синтаксиса, операционной семантики и правил типизации, которую мы произведем в главе 19.

18.1. Что такое объектно-ориентированное программирование?

Большинство споров на тему «В чем сущность такого-то явления?» скорее проявляют предрассудки их участников, чем проясняют какую-либо объективную истину, касающуюся темы спора. Попытки точного определения тер-

Примеры из этой главы являются термами простого типизированного лямбда-исчисления с подтипами (рис. 15.1), записями (15.3) и ссылками (13.1). Соответствующая реализация на OCaml называется `fullref`.

мина «объектно-ориентированный» в этом смысле не являются исключением. Тем не менее, можно отметить несколько основных конструкций, которые имеются в большинстве объектно-ориентированных языков, и которые все вместе служат основой для характерного стиля программирования с хорошо известными достоинствами и недостатками.

- 1. Множественные представления.** Вероятно, самая фундаментальная характеристика объектно-ориентированного стиля — это то, что когда по отношению к объекту применяется некоторая операция, объект сам определяет, какой код ее осуществляет. Два объекта с одним и тем же множеством операций (т. е., с одинаковым *интерфейсом*, interface) могут иметь совершенно разные внутренние представления, если каждый из них содержит реализацию операций, работающих с его конкретным представлением. Эти реализации называются *методами* (methods) объекта. Применение операции к объекту (*вызов метода*, method invocation) или, более красочно, *посылка объекту сообщения*, sending a message) приводит к поиску имени метода в таблице методов, связанной с объектом. Процесс поиска производится во время исполнения и называется *динамической диспетчеризацией методов* (dynamic dispatch).

В противоположность этому, обычный *абстрактный тип данных* (АТД) (abstract data type, ADT) состоит из набора значений, связанных с *единственной* реализацией операций над этими значениями. (Такое статическое определение реализаций имеет как достоинства, так и недостатки по сравнению с объектами; мы подробно рассмотрим этот вопрос в §24.2.)

- 2. Инкапсуляция.** Внутреннее представление объекта, как правило, недоступно за пределами его определения: только собственные методы объекта могут прямо считывать или изменять его поля.¹ Это означает, что изменения во внутреннем представлении объекта могут повлиять лишь на небольшой, ясно определенный участок программы; такое ограничение значительно облегчает чтение и поддержку больших систем.

¹ В некоторых объектно-ориентированных языках программирования, таких как Smalltalk, инкапсуляция обязательна — извне определения объекта невозможно даже *назвать имя* его полей. В других языках, скажем, C++ или Java, поля могут быть помечены как общедоступные (*public*) или приватные (*private*). С другой стороны, в Smalltalk все методы объекта публично доступны, тогда как C++ и Java позволяют отмечать *методы* модификатором *private*, который разрешает их вызов только из других методов того же объекта. В нашей книге мы эти детали опустим, но они подробно рассматривались в исследовательской литературе (Pierce and Turner, 1993; Fisher and Mitchell, 1998; Fisher, 1996a; Fisher and Mitchell, 1996; Fisher, 1996b; Fisher and Reppy, 1999).

Хотя в большинстве объектно-ориентированных языков инкапсуляция считается незаменимой возможностью, в нескольких языках это не так. *Мультиметоды* (multi-methods), имеющиеся в CLOS (Bobrow, DeMichiel, Gabriel, Keene, Kiczales, and Moon, 1988; Kiczales, des Rivières, and Bobrow, 1991), Cecil (Chambers, 1992, 1993), Dylan (Feinberg, Keene, Mathews, and Withington, 1997; Shalit) и KEA (Mugridge, Hamer, and Hosking, 1991), а также в лямбда-&-исчислении Кастаньи, Гелли и Лонго (Castagna, Ghelli, and Longo, 1995; Castagna, 1997) хранят состояние объектов отдельно от методов, и выбирают нужный вариант тела метода во время его вызова при помощи особых меток типа. Внутренние механизмы создания объектов, вызова методов, определения классов и т. п. в этих языках совершенно отличны от описываемых нами в этой главе, однако они приводят почти к таким же высокуюровневым идиомам программирования.

Абстрактные типы данных также допускают аналогичные формы инкапсуляции. Они делают конкретную форму представления типа видимой лишь внутри определенного участка программы (например, модуля, или определения АТД). Код вне этого участка может работать со значениями типа данных только с помощью операций, определенных внутри этой специальной области.

3. **Подтипы.** Тип объекта — его *интерфейс* (interface), — это просто множество имен и типов его операций. Внутреннее представление объекта не *фигурирует* в его типе, поскольку оно не влияет на набор действий, которые могут быть проделаны над объектом.

Интерфейсы объектов естественным образом определяют отношение подтиповирования. Если объект удовлетворяет интерфейсу I, то он, разумеется, удовлетворяет и вся кому интерфейсу J с более узким набором операций, чем I, поскольку всякий контекст, ожидающий J-объект, может вызывать у него только J-операции, и использование I-объекта в таком контексте является безопасным. (Таким образом, подтиповирование объектов подобно подтипованию записей. Более того, в модели объектов, разрабатываемой в этой главе, эти понятия будут равнозначными.) Способность игнорировать часть интерфейса объекта позволяет писать код, обрабатывающий многие различные виды объектов единообразно, требуя при этом только определенного общего набора операций.

4. **Наследование реализаций.** Объекты, в интерфейсах которых есть общая часть, часто будут иметь общность и в поведении, и нам хотелось бы эту общую часть поведения реализовать лишь однажды. В большинстве объектно-ориентированных языков такое повторное использование поведения достигается при помощи структур, которые называются *классами* (classes) — образцами, служащими для создания экземпляров объектов, — и механизма *подклассов/ subclassing*, позволяющего порождать новые классы из старых, добавляя реализации новых методов и по необходимости переопределяя реализации некоторых старых методов. (Некоторые объектно-ориентированные языки вместо классов используют механизм *делегатов* (delegates), сочетающий свойства объектов и классов.)
5. **Открытая рекурсия.** Еще одна удобная конструкция, имеющаяся в большинстве языков с объектами и классами, — возможность вызвать из тела одного метода другой метод того же объекта с помощью особой переменной, которая называется *self* (в некоторых языках — *this*). Особенность поведения *self* состоит в том, что она имеет *позднее связывание* late binding, что позволяет одному методу класса вызывать другой метод, который определяется позже, в некотором подклассе этого класса.

В оставшихся разделах этой главы последовательно вводятся эти конструкции. Вначале рассматриваются очень простые «самостоятельные» объекты, а затем описываются все более сложные варианты классов.

В последующих главах исследуются другие подходы к объектам и классам. Глава 19 описывает прямое определение (а не кодирование) объектов и

классов в стиле Java. В главе 27 мы возвращаемся к схеме кодирования, разработанной в этой главе, и повышаем эффективность конструирования классов при помощи ограниченной квантификации. В главе 32 разрабатывается более амбициозная версия схемы кодирования, работающая в чисто функциональном контексте.

18.2. Объекты

В простейшем варианте *объект* (object) — это просто структура данных, содержащая некоторое внутреннее *состояние* (state) и предоставляющая доступ к этому состоянию через набор *методов* (methods). Внутреннее состояние обычно состоит из нескольких изменяемых *переменных экземпляра* (instance variables) (или *полей*, fields), совместно используемых методами и недоступных остальным частям программы.

В этой главе нашим основным примером будут объекты, представляющие простые счетчики. Каждый объект-счетчик содержит одну числовую переменную и имеет два метода (т. е., реагирует на два сообщения) — `get`, заставляющий его вернуть текущее значение, и `inc`, увеличивающий это значение.

Простейший способ обеспечить это поведение при помощи конструкций, описанных нами в предыдущих главах, — использовать ссылочную ячейку в качестве внутреннего состояния объекта и запись, содержащую функции, в качестве методов. Объект-счетчик, текущее состояние которого равняется 1, выглядит так:

```
c = let x = ref 1 in
  {get = λ_:Unit. !x,
   inc = λ_:Unit. x:=succ(!x)};
▷ c : {get:Unit → Nat, inc:Unit → Unit}
```

Тела обоих методов записываются как функции с тривиальными параметрами (они записываются с помощью `_` (символа «подчерка»), поскольку нам не требуется обращаться к ним изнутри тел функций). Эти абстракции предотвращают вычисление тел методов в момент создания объекта, и позволяют позднее многократно вычислять эти тела, снова и снова применяя их к аргументу `unit`. Также обратите внимание, что состояние объекта совместно используется его методами и недоступно остальной программе: инкапсуляция состояния прямо возникает из лексической области видимости переменной `x`.

Чтобы вызвать метод объекта `c`, мы просто извлекаем поле из записи и применяем его к соответствующему аргументу. Например:

```
c.inc unit;
▷ unit : Unit
c.get unit
▷ 2 : Nat
(c.inc unit; c.inc unit; c.get unit)
```

```
> 4 : Nat
```

То, что метод `inc` возвращает `unit`, позволяет нам записать последовательность запросов на увеличение счетчика с помощью оператора «точка с запятой» (§11.3). Без него можно было бы записать последнюю строчку в эквивалентной форме:

```
let _ = c.inc unit in let _ = c.inc unit in c.get unit
```

Нам может потребоваться создать несколько счетчиков и работать с ними всеми. Для этого удобно будет ввести сокращенное именование их типа:

```
Counter = {get:Unit → Nat, inc:Unit → Unit}
```

В этой главе мы обращаем основное внимание на то, как объекты *устроены*, а не на то, как они *используются* для организации больших программ. Полезно, однако, показать хотя бы одну функцию, которая использует объекты, чтобы убедиться, что она правильно работает с объектами, у которых различаются внутренние представления. Вот простейший пример — функция, которая принимает объект-счетчик и трижды вызывает его метод `inc`:

```
inc3 = λc:Counter. (c.inc unit; c.inc unit; c.inc unit);

> inc3 : Counter → Unit
(inc3 c; c.get unit);

> 7 : Nat
```

18.3. Генераторы объектов

Мы узнали, как создать один объект-счетчик. Столь же просто написать *генератор счетчиков* — функцию, которая создает и возвращает новый счетчик при каждом вызове.

```
newCounter =
λ_:Unit. let x = ref 1 in
          {get = λ_:Unit. !x,
           inc = λ_:Unit. x:=succ(!x)};
```

```
> newCounter : Unit → Counter
```

18.4. Подтипы

Одна из причин популярности объектно-ориентированного стиля программирования заключается в том, что он позволяют обрабатывать различные объекты одним и тем же клиентским кодом. Предположим, например, что помимо определенных выше объектов типа `Counter` мы создаем также некоторое количество объектов, у которых есть дополнительный метод дляброса счетчика в начальное состояние (скажем, 1).

```
ResetCounter = {get:Unit → Nat, inc:Unit → Unit, reset:Unit →
Unit};

newResetCounter =
λ_:Unit. let x = ref 1 in
  {get = λ_:Unit. !x,
   inc = λ_:Unit. x:=succ(!x),
   reset = λ_:Unit. x:=1};

▷ newResetCounter : Unit → ResetCounter
```

Так как тип `ResetCounter` содержит все поля типа `Counter` (плюс еще одно), то по правилу подтиповирования для записей получаем, что `ResetCounter <: Counter`. Это означает, что клиентские функции вроде `inc3`, принимающие в качестве аргументов счетчики, безопасно использовать и со сбрасываемыми счетчиками.

```
rc = newResetCounter unit;

▷ rc : ResetCounter

(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);

▷ 4 : Nat
```

18.5. Группировка переменных экземпляра

До сих пор состоянием во всех наших объектах служила всего одна ссылочная ячейка. Разумеется, в более интересных объектах часто будет по несколько переменных экземпляра. В дальнейшем удобно будет иметь возможность обращаться со всеми этими переменными как с единым целым. Чтобы обеспечить это, изменим внутреннее представление счетчиков: сделаем его *записью*, которая содержит ссылочные ячейки, и будем обращаться к переменным экземпляра в телах методов через проекцию полей этой записи.

```
c = let r = {x=ref 1} in
  {get = λ_:Unit. !(r.x),
   inc = λ_:Unit. r.x := succ(!r.x)};
```

```
▷ c : Counter
```

Тип этой записи, содержащей переменные экземпляра, называется *типом представления* (representation type) объекта.

```
CounterRep = {x: Ref Nat};
```

18.6. Простые классы

Определения типов `newCounter` и `newResetCounter` одинаковы, за исключением метода `reset` во втором из них. Разумеется, оба эти определения столь

кратки, что повторение не создает особых трудностей, но, если представить себе, что эти определения занимают много страниц, как это часто случается на практике, то становится ясно, что было бы полезно уметь описывать общую функциональность в одном месте. Механизм, позволяющий это делать, в большинстве объектно-ориентированных языков называется *классами* (classes).

Механизмы определения классов в настоящих объектно-ориентированных языках, как правило, очень сложны и перегружены деталями — `self`, `super`, аннотации видимости, статические поля и методы, внутренние классы, аннотации вроде `final` и `Serializable` и т. д. и т. п.² На большую часть этих деталей мы пока не будем обращать внимания, и сосредоточимся на самых базовых свойствах классов: повторное использование кода через наследование реализаций, а также позднее связывание переменной `self`. Начнем с первого вопроса.

В своей простейшей форме класс — это просто структура данных, которая содержит набор методов. У этой структуры можно *создать экземпляр* (`instantiate`), получая при этом новый объект, а также ее можно *расширить* (`extend`), получив при этом новый класс.

Почему мы не можем повторно использовать методы какого-либо *объекта*-счетчика, если хотим создать сбрасываемый счетчик? Да потому, что в каждом конкретном объекте-счетчике тела методов содержат ссылки на какую-то конкретную запись с переменными экземпляра. Ясно, что если мы хотим использовать тот же самый код с другой записью переменных экземпляра, нам требуется *абстрагировать* (`abstract`) методы по отношению к переменным экземпляра. Это приводит к разбиению функции `newCounter` на две части, одна из которых определяет тела методов относительно *произвольной* записи переменных экземпляра,

```
counterClass =
  λr:CounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x = succ(!(r.x))};
> counterClass : CounterRep → Counter
```

а вторая создаёт новую запись переменных экземпляра и передает ее телам методов, получая в результате объект:

```
newCounter =
  λ_:Unit. let r = {x=ref 1} in
            counterClass r;
> newCounter : Unit → Counter
```

Тела методов из `counterClass` можно использовать для определения новых классов, называемых *подклассами* (subclasses). Можно, например, определить класс сбрасываемых счетчиков:

²Основная причина этой сложности в том, что в большинстве этих языков классы служат единственным механизмом структуризации программ в целом. Действительно, есть только один широко используемый язык — OCaml, где имеются как классы, так и мощная система модулей. Из-за этого в большинстве языков классы превращаются в свалку всех языковых конструкций, имеющих какое-то отношение к структурированию крупных программ.

```

resetCounterClass =
  λr: CounterRep.
    let super = counterClass r in
      {get    = super.get,
       inc   = super.inc,
       reset = λ_:Unit. r.x:= 1};
▷ resetCounterClass : CounterRep → ResetCounter

```

Как и `counterClass`, эта функция принимает запись переменных экземпляра и возвращает объект. Внутри она сначала создает при помощи `counterClass` объект-счетчик с *той же самой* записью переменных экземпляра `r` (этот «родительский объект» привязан к переменной `super`). Затем она создаёт новый объект, копируя поля `get` и `inc` из переменной `super`, а в качестве значения поля `reset` передает новую функцию. Так как `super` основан на представлении `r`, все три метода совместно используют одни и те же переменные экземпляра.

Чтобы создать новый объект-сбрасываемый счетчик, мы, опять же, просто выделяем память для его переменных экземпляра и вызываем `resetCounterClass`, который и проделывает основную работу.

```

newResetCounter =
  λ_:Unit. let r = {x=ref 1} in resetCounterClass r;
▷ newResetCounter : Unit → ResetCounter

```

Упражнение 18.6.1 [РЕКОМЕНДУЕТСЯ, ★★]: *Напишите подкласс `resetCounterClass`, который имеет дополнительный метод `dec`, уменьшающий на единицу текущее значение счетчика. Проверьте свой новый класс при помощи интерпретатора `fullref`.*

Упражнение 18.6.2 [★★ →]: *Явное копирование большинства полей надкласса в запись методов подкласса выглядит достаточно неуклюже — при этом мы устранием повторение текста методов надкласса в подклассе, но все равно требуется набрать много текста. Если бы мы собирались разрабатывать в этом стиле большие объектно-ориентированные программы, нам очень скоро захотелось бы иметь конструкцию вроде `super with {reset = λ_:Unit. r.x:=1}`, означающую «запись, такая же, как `super`, только с дополнительным полем `reset`, содержащим функцию `λ_:Unit. r.x:=1`». Запишите синтаксис, операционную семантику и правила типизации для этой конструкции.*

Нужно особо подчеркнуть, что классы являются *значениями*, а не *типами*. Кроме того, если потребуется, можно создать множество классов, порождающих объекты одного и того же типа. В популярных объектно-ориентированных языках вроде C++ и Java статус классов более сложен — они используются и как типы во время компиляции, и как структуры данных во время исполнения. Этот вопрос подробнее рассматривается в §19.3.

18.7. Добавление новых переменных экземпляра

Так получилось, что у обоих наших объектов, счетчика и сбрасываемого счетчика, представление одинаково. Однако в общем случае подклассу может потребоваться расширить не только список методов, но и список переменных по сравнению с надклассом, от которого он произведен. Допустим, например, что мы хотим определить класс «счетчиков с восстановлением», у которых метод `reset` восстанавливает состояние не к заранее заданной константе, а к тому значению, которое объект имел на момент последнего вызова метода `backup`:

```
BackupCounter = {get:Unit → Nat, inc:Unit → Unit,
                 reset:Unit → Unit, backup:Unit → Unit};
```

Чтобы реализовать такие счетчики, нужна дополнительная переменная экземпляра, в которой будет храниться копия значения состояния:

```
BackupCounterRep = {x:Ref Nat, b: Ref Nat};
```

Так же, как мы произвели `resetCounterClass` из `counterClass`, скопировав методы `get` и `inc` и добавив `reset`, произведем теперь `backupCounterClass` из `resetCounterClass`, копируя `get` и `inc` и добавляя новое тело `reset` и `backup`.

```
backupCounterClass =
  λr:BackupCounterRep.
    let super = resetCounterClass r in
      {get      = super.get,
       inc     = super.inc,
       reset   = λ_:Unit. r.x:=!(r.b),
       backup  = λ_:Unit. r.b:=!(r.x)};
```

```
▷ backupCounterClass : backupCounterRep → BackupCounter
```

В этом определении интересны две подробности. Во-первых, несмотря на то, что в родительском объекте `super` имеется метод `reset`, мы пишем новую реализацию, поскольку нам нужно другое поведение этого метода. Новый класс *переопределяет* (*override*) метод `reset` своего надкласса. Во-вторых, при типизации выражения, которое строит `super`, существенным образом используется подтиповование: `resetCounterClass` ожидает аргумента типа `CounterRep`, который является надтипом `BackupCounterRep` — типа, который на самом деле имеет аргумент `r`. Другими словами, мы передаем родительскому объекту запись переменных экземпляра с большим количеством полей, чем требуется его методам.

Упражнение 18.7.1 [РЕКОМЕНДУЕТСЯ, ★★]: *Определите подкласс класса `backupCounterClass` с двумя новыми методами, `reset2` и `backup2`, управляющими вторым «резервным регистром». Этот регистр должен быть совершенно независим от регистра, добавленного в `backupCounterClass`: вызов*

`reset` должен восстанавливать состояние счетчика к значению, имевшемуся при последнем вызове `backup` (как это и сейчас происходит), а вызов `reset2` должен восстанавливать состояние счетчика к значению, имевшемуся во время последнего вызова `backup2`. Проверьте свой класс с помощью интерпретатора `fullref`.

18.8. Вызов методов надкласса

С помощью переменной `super` мы копировали функциональность из надклассов в новые подклассы. Кроме того, мы можем использовать `super` в телях определений методов, чтобы *расширять* (extend) поведение надкласса чем-то новым. Предположим, например, что нам требуется вариант класса `backupCounter`, в котором каждый вызов метода `inc` предваряется вызовом `backup`. (Совершенно непонятно, зачем бы такое могло понадобиться на практике — это всего лишь пример.)

```
funnyBackupCounterClass =
  λr: BackupCounterRep .
    let super = backupCounterClass r in
      {get = super.get,
       inc = λ_:Unit. (super.backup unit; super.inc unit),
       reset = super.reset,
       backup = super.backup};

▷ funnyBackupCounterClass : BackupCounterRep → BackupCounter
```

Обратите внимание, как с помощью вызовов `super.inc` и `super.backup` в новом определении `inc` мы избегаем повторения кода надкласса для `inc` и `backup`. В примерах большего размера благодаря повторному использованию функциональности выигрыш в таких ситуациях может быть очень значительным.

18.9. Классы с переменной `self`

Последнее наше расширение состоит в том, чтобы позволить методам класса ссылаться друг на друга через переменную `self`. Для чего нужно это расширение? Предположим, что мы хотим реализовать класс счетчиков с методом `set`, с помощью которого можно извне устанавливать значение счетчика в определенное число.

```
SetCounter = {get:Unit → Nat, set:Nat → Unit, inc:Unit →
Unit};
```

Кроме того, допустим, что мы хотим написать метод `inc` в терминах `set` и `get`, а не через прямое считывание и присваивание переменной экземпляра `x`. (Представьте себе огромную систему, в которой определения `set` и `get` занимают много страниц.) Поскольку `get`, `set` и `inc` определены в рамках одного и того же класса, мы, в сущности, хотим сделать методы класса взаимно рекурсивными.

Мы уже видели в §11.11, как взаимно рекурсивные записи или функции строятся с помощью оператора `fix`. Мы просто абстрагируем запись методов через параметр, который сам является записью функций (мы эту запись называем `self`), а затем с помощью оператора `fix` «затягиваем узел», чтобы запись, создаваемая нами, сама передавалась как `self`.

```
setCounterClass =
  λr:CounterRep.
    fix
      (λself: SetCounter.
        {get = λ_:Unit. !(r.x),
         set = λi:Nat. r.x:=i,
         inc = λ_:Unit. self.set (succ (self.get unit)))};

▷ setCounterClass : CounterRep → SetCounter
```

У этого класса нет родителя, так что переменная `super` не требуется. Вместо этого, тело метода `inc` вызывает `get`, а затем `set` из записи с методами, переданной ему в качестве параметра `self`. Это использование `fix` заметно только изнутри реализации `setCounterClass`. Счетчики с присваиванием можно создавать как обычно.

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
            setCounterClass r;

▷ newSetCounter : Unit → SetCounter
```

18.10. Открытая рекурсия через `self`

В большинстве объектно-ориентированных языков поддерживается более общая форма рекурсивных вызовов между методами, известная под названием *рекурсии общего вида* (general recursion), или *позднего связывания* (late binding) переменной `self`. Можно добиться этой более общей формы поведения, избавившись от использования `fix` в определении класса,

```
setCounterClass =
  λr:CounterRep.
    λself: SetCounter.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat. r.x:=i.
       inc = λ_:Unit. self.set (succ(self.get unit))};

▷ setCounterClass : CounterRep → SetCounter → SetCounter
```

и поместив его вместо этого в функцию создания объекта.

```
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
            fix (setCounterClass r);
▷ newSetCounter : Unit → SetCounter
```

Заметим, что перенос `fix` изменяет тип `SetCounterClass`: теперь он абстрагируется не только от записи с переменными экземпляра, но и от «объекта `self`». И тот, и другой параметр передаются ему в момент создания экземпляра.

Открытая рекурсия через `self` интересна потому, что она позволяет методам *надкласса* вызывать методы *подкласса*, несмотря на то, что подкласс еще не существует в тот момент, когда определяется надкласс. В сущности, мы изменили интерпретацию переменной `self` так, что вместо «методов текущего класса» она дает доступ к «методам класса, экземпляром которого является текущий объект [а это может быть подкласс текущего класса]».

Например, предположим, что нам нужно построить подкласс наших счетчиков с присваиванием так, чтобы проводился подсчет вызовов метода `set`. Интерфейс этого класса содержит дополнительную операцию для доступа к счетчику доступа,

```
InstrCounter = {get:Unit → Nat, set:Nat → Unit,
                inc:Unit → Unit, accesses:Unit → Nat};
```

а представление содержит переменную экземпляра, в которой хранится счетчик доступа:

```
InstrCounterRep = {x: Ref Nat, a: Ref Nat};
```

В определении измененного класса счетчиков методы `inc` и `get` копируются из определенного выше `setCounterClass`. Метод `accesses` пишется обычным способом. В методе `set` мы сначала увеличиваем счетчик доступа, а затем через `super` вызываем метод `set` надкласса.

```
instrCounterClass =
  λr: InstrCounterRep.
    λself: InstrCounter.
      let super = setCounterClass r self in
        {get = super.get,
         set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
         inc = super.inc,
         accesses = λ_:Unit. !(r.a)};
  ▷ instrCounterClass : InstrCounterRep →
                           InstrCounter → InstrCounter
```

Из-за открытой рекурсии через `self`, вызов `set` в теле `inc` приведет к увеличению значения переменной экземпляра `a`, несмотря на то, что эта часть поведения `set` определена в подклассе, а определение `inc` находится в надклассе.

18.11. Открытая рекурсия и порядок вычислений

У нашего определения `InstrCounterClass` есть только одна проблема — его невозможно использовать для создания экземпляров! Если мы напишем `newInstrCounter` обычным способом

```

newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
            fix (instrCounterClass r);

▷ newInstrCounter : Unit → InstrCounter

```

а затем попытаемся создать счетчик с подсчетом доступа, применив эту функцию к `Unit`,

```
ic = newInstrCounter unit;
```

то вычисление этого выражения зациклится. Чтобы понять, почему это так, рассмотрим последовательность шагов вычисления, выполняемых при запуске этого терма.

1. Сначала мы применяем `newInstrCounter` к `unit`, получая

```
let r = {x=ref 1, a=ref 0} in fix (instrCounterClass r)
```

2. Затем мы выделяем две ссылочных ячейки, упаковываем их в виде записи (назовем ее `<ivars>`) и подставляем `<ivars>` вместо переменной `r` в оставшейся части терма.

```
fix (instrCounterClass <ivars>)
```

3. Передаем `<ivars>` в `instrCounterClass`. Поскольку `instrCounterClass` начинается с двух лямбда-абстракций, мы немедленно получаем функцию, ожидающую значения `self`,

```
fix (λself:InstrCounter.
```

```
    let super = setCounterClass <ivars> self in <imethods>)
```

где `<imethods>` — запись, содержащая методы класса счетчиков с подсчетом доступа. Назовем эту функцию `<f>`, и запишем текущее состояние в виде (`fix <f>`).

4. Применяем правило вычисления для `fix` (E-FIX по рис. 11.12, с. 167). Это правило «разворачивает» `fix <f>`, подставляя (`fix <f>`) вместо переменной `self` в теле `<f>`. Получаем

```
let super = setCounterClass <ivars> (fix <f>) in <imethods>
```

5. Теперь мы редуцируем применение `setCounterClass` к `<ivars>` и получаем

```
let super = (λself:SetCounter. <smethods>) (fix <f>)
           in <imethods>
```

где `<smethods>` — запись, состоящая из методов счетчика с присваиванием.

6. По правилам вычисления для термов-применений, мы не можем редуцировать применение `(λself:SetCounter. <smethods>)` к `(fix <f>)`, пока терм `(fix <f>)` не сведен к значению. Поэтому на следующем шаге вычисления мы снова разворачиваем `fix <f>`, получая

```

let super = ( $\lambda self: SetCounter.$  <smethods>
             (let super = setCounterClass <iVars> (fix <f>)
              in <iMethods>)
            in <iMethods>)

```

7. Поскольку аргумент внешней лямбда-абстракции по-прежнему не является значением, мы должны продолжать вычисление внутренней абстракции. Применяем `setCounterClass` к `<iVars>` и получаем

```

let super = ( $\lambda self: SetCounter.$  <smethods>
             (let super = ( $\lambda self: SetCounter.$  <smethods>
                           (fix <f>)
                          in <iMethods>)
            in <iMethods>)

```

8. Теперь у нас получился внутренний терм-применение, подобный внешнему. Как и раньше, этот внутренний терм не может быть редуцирован, пока полностью не вычислен его аргумент, `fix <f>`. Таким образом, на следующем шаге мы снова развернем `fix <f>`, получая еще более глубоко вложенное выражение той же формы, что и на шаге 6. Теперь должно быть понятно, что мы *никогда* не доберемся до вычисления внешнего терма-применения.

Интуитивно ясно, что проблема возникает из-за того, что аргумент оператора `fix` слишком рано использует свой собственный аргумент `self`. Операционная семантика `fix` определена в предположении, что когда мы применяем `fix` к некоторой функции $\lambda x. t$, переменная x должна встречаться в теле t только в «зашщищенных» позициях, таких как тела вложенных лямбда-абстракций. Например, на с. 161 мы определили `iseven` путем применения `fix` к функции вида $\lambda ie. \lambda x. \dots$, где рекурсивное упоминание `ie` было защищено абстракцией по x . Определение `instrCounterClass`, напротив, пытается немедленно использовать `self` при вычислении значения `super`.

Мы можем пытаться преодолеть эту сложность несколькими способами:

- Можно защитить ссылку на `self` в теле `instrCounterClass`, избежав таким образом ее слишком раннего вычисления. Например, можно вставить абстракцию-«заглушку» специально для этой цели. Мы рассматриваем это решение ниже. Как мы увидим, оно не вполне удовлетворительно, но его легко описать и понять с помощью уже рассмотренных нами механизмов. Это решение также окажется полезным позднее, когда мы рассмотрим чисто функциональное представление объектов в главе 32.
- Можно попытаться найти иные способы моделирования семантики классов на основе низкоуровневых конструкций языка. Например, вместо того, чтобы создавать таблицу методов класса при помощи `fix`, можно построить ее более явным образом, через ссылки. Мы развиваем эту идею в §18.12, и ещё раз — в главе 27.
- Можно забыть о представлении объектов и классов в терминах лямбда-абстракций, записей и `fix`, а вместо этого рассматривать их как элементарные конструкции языка, с собственными правилами вычисления

(и типизации). В этом случае мы можем просто выбрать такие правила вычисления, которые точно соответствуют нашим требованиям к поведению объектов и классов, а не пытаться обойти проблемы с существующими правилами для применения `fix`. Этот подход обсуждается в главе 19.

Использование абстракций-заглушек для управления порядком исполнения — прием, хорошо известный в сообществе функциональных программистов. Идея состоит в том, что произвольное выражение `t` можно превратить в функцию-обёртку $\lambda : \text{Unit}. t$, которая называется *thunk*. Такая «обёртка» `t` с точки зрения синтаксиса является значением; все действия, связанные с вычислением `t`, откладываются до тех пор, пока функция-обёртка не будет применена к аргументу `unit`. Таким образом, мы можем работать с `t` в невычисленном виде, а затем, позже, запросить его результат.

Сейчас нам требуется задержать вычисление `self`. Мы добиваемся этого, сменив его тип с объекта (например, `SetCounter`) на функцию-обёртку, содержащую объект $(\text{Unit} \rightarrow \text{SetCounter})$. При этом надо а) изменить тип параметра `self`, передаваемого в класс, б) добавить абстракцию-заглушку перед тем, как вернуть объект-результат, и в) заменить каждое вхождение `self` в телах методов на `(self unit)`.

```
setCounterClass =
  λr: CounterRep.
  λself: Unit → SetCounter.
  λ_: Unit.
    {get = λ_: Unit. !(r.x),
     set = λi: Nat. r.x := i,
     inc = λ_: Unit. (self unit).set(succ((self unit).get unit))}
> setCounterClass : CounterRep →
                    (Unit → SetCounter) → Unit → SetCounter
```

Поскольку мы не хотим изменять тип `newSetCounter` (эта функция по-прежнему должна возвращать объект), нам нужно немного модифицировать ее определение, чтобы она передавала аргумент `unit` в функцию-обёртку, которая получается при создании неподвижной точки `setCounterClass`.

```
newSetCounter =
  λ_: Unit. let r = {x=ref 1} in
             fix (setCounterClass r) unit;

> newSetCounter : Unit → SetCounter
```

Аналогичные изменения требуются и в определении `instrCounterClass`. Заметим, что эти изменения не требуют от нас никаких творческих способностей: как только мы изменили тип `self`, все остальное вытекает из правил типизации.

```
instrCounterClass =
  λr: InstrCounterRep.
  λself: Unit → InstrCounter.
  λ_: Unit.
    let super = setCounterClass r self unit in
```

```

    {get = super.get,
     set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
     inc = super.inc,
     accesses = λ_:Unit. !(r.a)};
```

▷ instrCounterClass : InstrCounterRep →
 (Unit → InstrCounter) → Unit →
 InstrCounter

Наконец, мы модифицируем newInstrCounter так, чтобы он передавал аргумент-заглушку в функцию-обёртку, порожденную конструкцией fix.

```

newInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0} in
            fix (instrCounterClass r) unit;

▷ newInstrCounter : Unit → InstrCounter
```

Теперь с помощью newInstrCounter мы можем создать объект.

```

ic = newInstrCounter unit;

▷ ic : InstrCounter
```

Напомним, что именно попытка создать объект до добавления функций-обёрток и не могла завершиться в предыдущей версии кода.

Следующие тесты показывают, что метод accesses подсчитывает обращения и к set, и к inc, как мы и добивались.

```

(ic.set 5; ic.accesses unit);

▷ 1 : Nat

(ic.inc unit; ic.get unit);

▷ 6 : Nat

ic.accesses unit;

▷ 2 : Nat
```

Упражнение 18.11.1 [РЕКОМЕНДУЕТСЯ, ★★★]: При помощи интерпретатора fullref реализуйте следующие расширения к имеющимся у нас классам:

1. Перепишите instrCounterClass так, чтобы он также подсчитывал и обращения к get.
2. Расширьте измененный instrCounterClass подклассом, который добавляет метод reset, как в §18.4.
3. Добавьте еще один подкласс, который поддерживает еще и резервные значения (метод backup, §18.7).

18.12. Более эффективная реализация

Тесты в конце предыдущего раздела показывают, что наша реализация классов осуществляет «открытую рекурсию» при вызове методов через `self`, как это принято в таких языках, как Smalltalk, C++ или Java. Следует, однако, заметить, что реализация эта не вполне удовлетворительна с точки зрения эффективности. Все функции-обёртки, вставленные нами, чтобы заставить вычисление `fix` завершаться, откладывают вычисление таблиц методов для классов. Заметим, в частности, что все вызовы `self` в телах методов превращаются в `(self unit)`, — то есть, методы переменной `self` вычисляются заново при каждом рекурсивном вызове любого из них!

Этих повторных вычислений можно избежать, если для того, чтобы «заявлять узел» в иерархии классов при построении объектов, воспользоваться не неподвижными точками, а ссылочными ячейками.³ Вместо того, чтобы строить абстракцию классов на записи с методами, которая называется `self` и которая впоследствии будет создана при помощи `fix`, мы строим абстракцию на *ссылке* на запись с методами, и создаем эту запись *с самого начала*. А именно, мы строим экземпляр класса, выделяя ячейку для его методов в памяти (и присваивая ей значение-заглушку), затем строим настоящие методы (и передаем им указатель на ячейку памяти, через которую они делают рекурсивные вызовы), и, наконец, меняем значение в ячейке, чтобы она содержала настоящие методы. Вот, например, снова `setCounterClass`:

```
setCounterClass =
  λr:CounterRep. λself: Ref SetCounter.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self).set (succ ((!self).get unit))};

▷ setCounterClass : CounterRep → (Ref SetCounter) → SetCounter
```

Параметр `self` — указатель на ячейку, которая содержит методы текущего объекта. При вызове `setCounterClass` эта ячейка инициализируется значением-заглушкой:

```
dummySetCounter =
  {get = λ_:Unit. 0,
   set = λi:Nat. unit,
   inc = λ_:Unit. unit};

▷ dummySetCounter : SetCounter

newSetCounter =
  λ_:Unit.
    let r = {x=ref 1} in
    let cAux = ref dummySetCounter in
      (cAux := (setCounterClass r cAux); !cAux);
```

³ В сущности, эту же идею мы использовали при решении упражнения 13.5.8. Я благодарен Джеймсу Рили за мысль о том, что этот прием можно применить при конструировании классов, используя ковариантность типов `Source`.

```
> newSetCounter : Unit → SetCounter
```

Однако поскольку все операции разыменования (`!self`) защищены лямбда-абстракциями, никаких обращений к ячейке не будет до тех пор, пока функция `newSetCounter` не запишет в неё настоящее значение.

Чтобы обеспечить возможность создания подклассов `setCounterClass`, требуется еще немного уточнить его тип. Каждый класс ожидает, что его параметр `self` будет иметь такой же тип, как запись методов, которую он создаёт. То есть, если мы определяем подкласс счетчиков с подсчетом доступа, параметр `self` в этом классе будет указателем на запись с методами счетчиков с подсчетом доступа. Однако, как мы видели в §15.5, типы `Ref SetCounter` и `Ref InstrCounter` несовместимы — невозможно продвинуть второй из них и получить первый. Это приводит к ошибке (т. е., несоответствию типа параметра) при попытке создания `super` в определении `instrCounterClass`.

```
instrCounterClass =
  λr:instrCounterRep. λself: Ref InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit. !(r.a)};
```

```
> Error: parameter type mismatch
```

Разрешить это затруднение можно, заменив конструктор `Ref` в типе `self` на `Source` — т. е. передав в класс только разрешение читать значение по ссылке на методы, но не разрешение туда писать (оно классу все равно не нужно). Как мы видели в §15.5, конструктор `Source` разрешает ковариантное подтиповирование — т. е. выполняется условие `Source InstrCounter <: Source SetCounter`, поэтому создание `super` в `instrCounterClass` будет правильно типизировано.

```
setCounterClass =
  λr:CounterRep. λself: Source SetCounter.
  {get = λ_:Unit. !(r.x),
   set = λi:Nat. r.x:=i,
   inc = λ_:Unit. (!self).set (succ ((!self).get unit))};
```

```
> setCounterClass : CounterRep → (Source SetCounter) →
  SetCounter
```

```
instrCounterClass =
  λr:InstrCounterRep. λself: Source InstrCounter.
  let super = setCounterClass r self in
  {get = super.get,
   set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = λ_:Unit. !(r.a)};
```

```
> instrCounterClass : InstrCounterRep →
```

(Source InstrCounter) → InstrCounter

Чтобы построить объект-счетчик с подсчетом доступа, мы сначала, как и раньше, определяем заглушку-набор методов такого счетчика в качестве начального значения указателя `self`.

```
dummyInstrCounter =
  {get = λ_:Unit. 0,
   set = λi:Nat. unit,
   inc = λ_:Unit. unit,
   accesses = λ_:Unit. 0};
```

```
> dummyInstrCounter : InstrCounter
```

Затем мы создаем объект, выделяя в памяти место для переменных экземпляра и методов, вызываем `instrCounterClass`, чтобы создать настоящие методы, и заменяем значение в ссылочной ячейке.

```
newInstrCounter =
  λ_:Unit.
    let r = {x=ref 1, a=ref 0} in
    let cAux = ref dummyInstrCounter in
      (cAux := (instrCounterClass r cAux); !cAux);
```

```
> newInstrCounter : Unit → InstrCounter
```

Код для построения таблицы методов (в `instrCounterClass` и в `setCounterClass`) теперь вызывается один раз при создании объекта, а не при каждом вызове метода. Таким образом, мы добиваемся своей цели, однако, по-прежнему, не столь эффективно, как нам того хотелось бы: ведь таблица методов, которую мы конструируем для каждого объекта-счетчика с подсчетом доступа, всегда одна и та же, так что, казалось бы, мы могли бы построить ее *один раз* при определении класса и больше не трогать. В главе 27 мы увидим, как добиться этого при помощи ограниченной квантификации (которая описана в главе 26).

18.13. Резюме

В первом разделе этой главы были перечислены несколько отличительных особенностей объектно-ориентированного стиля. Вспомним эти характеристики и кратко обсудим, как они соотносятся с примерами, представленными в этой главе.

- 1. Множественные представления.** Все объекты, рассмотренные нами в этой главе — счетчики, т. е., принадлежат к типу `Counter`. Однако их представления сильно различаются: от единственной ссылочной ячейки в §18.2 до записи, содержащей несколько ссылок, в §18.9. Каждый объект представляет собой запись, состоящую из функций, которые служат для реализации методов типа `Counter` (и, возможно, каких-то еще), соответствующей внутреннему представлению этого объекта.

2. **Инкапсуляция.** То, что переменные экземпляра каждого объекта доступны только внутри его методов, непосредственно следует из нашего способа построения объектов — через передачу записи переменных экземпляра в функцию-конструктор, создающую методы. Очевидно, что *имена* переменных экземпляра можно увидеть только изнутри методов
3. **Подтипование** между типами объектов в нашей реализации — это просто обыкновенное подтипование между типами записей, состоящих из функций.
4. **Наследование реализаций.** Мы моделировали наследование реализаций через копирование реализаций методов из существующего надкласса в новый подкласс. При этом возникает несколько интересных технических деталей: строго говоря, как надкласс, так и новый подкласс представляют собой *функции* из переменных экземпляра в запись, состоящую из методов. Подкласс принимает свои переменные экземпляра в качестве параметра, затем создает экземпляр надкласса с полученными переменными и получает запись методов надкласса, работающих с теми же самыми переменными.
5. **Открытая рекурсия.** Открытая рекурсия, обеспечивающая в настоящих объектно-ориентированных языках через переменную `self` (или `this`), моделируется нами через абстракцию не только по переменным экземпляра, но и по параметру `self`, который методы могут использовать для обращения к другим методам того же объекта. Этот параметр получает значение при создании объекта, когда мы «завязываем узел» при помощи `fix`.

Упражнение 18.13.1 [★★★]: Еще одно свойство объектов, которое часто оказывается полезным, — понятие тождества объектов (*object identity*), операция `sameObject`, которая возвращает `true` в том случае, если оба ее аргумента при вычислении дают один и тот же объект, и `false`, если они имеют значением объекты, созданные в разное время (различными вызовами функций `new`). Как можно расширить модель объектов из этой главы, чтобы ввести поддержку тождества объектов?

18.14. Дополнительные замечания

Представление объектов — неиссякаемый источник примеров и задач для сообщества исследователей языков программирования. Одно из первых решений было дано Рейнольдсом (Reynolds, 1978); всеобщий интерес к этой задаче пробудила статья Карделли (Cardelli, 1984). Понятие о `self` как о неподвижной точке было разработано Куком (Cook, 1989), Куком и Палсбергом (Cook and Palsberg, 1989), Камином (Kamin, 1988) и Редди (Reddy, 1988); связи между этими моделями исследовались Камином и Редди (Kamin and Reddy, 1994), а также Брюсом (Bruce, 1991).

Некоторые важные ранние работы в этой области собраны в книге Gunter and Mitchell, 1994. Обзор более поздних достижений можно найти у Брюса, Карделли и Пирса (Bruce, Cardelli, and Pierce, 1999), а также у Абади и

Карделли (Abadi and Cardelli, 1996). Брюс (Bruce, 2002) дает современную картину исследований в этой области. Альтернативные подходы к основам объектов и их систем типов можно найти у Палсберга и Шварцбаха (Palsberg and Schwartzbach, 1994) и у Кастаньи (Castagna, 1997).

Дополнительные исторические замечания можно найти в конце главы 32.

Наследованию реализаций
придается чрезмерно большое
значение.

Гради Буч

Глава 19

Расширенный пример: Облегченная Java

В гл. 18 мы видели, как с помощью лямбда-исчисления с подтипами, записями и ссылками можно смоделировать некоторые ключевые конструкции объектно-ориентированного программирования. Нашей целью было добиться более глубокого понимания этих конструкций путем их *кодирования* (encoding) в терминах более простых элементов языка. В этой главе мы используем другой подход и покажем, как адаптировать идеи из предыдущих глав, чтобы *напрямую* описать базовый объектно-ориентированный язык на основе Java. Предполагается, что читатель уже знаком с Java.

19.1. Введение

Формальное моделирование может оказать большую помощь в проектировании сложных искусственных объектов, таких как языки программирования. С помощью формальной модели можно точно описать некоторый аспект проектируемого объекта, сформулировать и доказать его свойства, и обратить внимание на проблемы, которые в другом случае остались бы незамеченными. Однако при построении модели всегда налицоует противоречие между полнотой и компактностью: чем больше аспектов объекта модель отражает, тем более громоздкой она оказывается. Часто имеет смысл предпочесть модель менее полную, но более компактную, и получить максимальную отдачу при минимальных затратах. Эту стратегию можно наблюдать во множестве недавних статей по формальным свойствам Java, в которых игнорируются такие сложные черты языка, как параллелизм и рефлексия, а основное внимание концентрируется на тех фрагментах языка, где можно применить хорошо изученные теоретические механизмы.

Примеры в этой главе являются термами Облегченной Java (рис. 19.1–19.4). Соответствующей реализации на OCaml не существует, но, поскольку Облегченная Java спроектирована как строгое подмножество Java, любая реализация Java годится для выполнения примеров.

Облегченная Java (Featherweight Java, FJ) была предложена в статье Игараси, Пирса и Уодлера (Igarashi, Pierce, and Wadler, 1999) в качестве одного из вариантов *минимального ядра* (minimal core), на основе которого можно смоделировать систему типов Java. FJ маниакально придерживается принципа компактности, отдавая ему предпочтение перед полнотой. В этом языке остается только пять видов термов: создание объекта, вызов метода, доступ к полю, приведение типа и переменная. Синтаксис, правила типизации и операционная семантика языка легко умещаются на одной странице формата А4. В сущности, целью проекта было отбросить как можно больше конструкций (включая присваивание), но сохранить основные черты типизации в Java. Существует простое соответствие между FJ и чисто функциональным ядром Java, в том смысле, что всякая программа на FJ является буквально еще и программой на Java.

FJ лишь ненамного превышает по размеру лямбда-исчисление или объектное исчисление Абади и Карделли (Abadi and Cardelli, 1996). FJ намного меньше, чем другие формальные модели Java-подобных языков, основанных на классах, в том числе модели Дроссопулу, Айзенбаха и Хуршида (Drossopoulou, Eisenbach, and Khurshid, 1999), Сайма (Syme, 1997), Нипкова и Охаймба (Nipkow and von Oheimb, 1998), а также Флатта, Кришнамурти и Феллесена (Flatt, Krishnamurthi, and Felleisen, 1998a,b). Благодаря своему размеру, FJ может концентрироваться только на нескольких ключевых вопросах. Например, как мы увидим, приведение типа в Java описывается сложнее, чем можно было бы предположить.

Основное применение FJ — моделирование расширений Java. Поскольку сама FJ очень мала, в её рамках можно уделить все внимание ключевым аспектам конкретного расширения. Более того, поскольку доказательство типовой безопасности FJ весьма просто, можно построить строгое доказательство даже для довольно большого расширения. Первая же статья о FJ продемонстрировала это свойство, добавив в FJ обобщенные классы и методы по образцу GJ (Bracha, Odersky, Stoutamire, and Wadler, 1998). В следующей статье (Igarashi, Pierce, and Wadler, 2001) было построено формальное описание *сырых* (raw) типов, введенных в GJ, чтобы облегчить переписывание Java-программ на GJ. Игараси и Пирс (Igarashi and Pierce, 2000) использовали FJ как основу для изучения свойств *внутренних классов* (inner classes) в Java. Кроме того, FJ использовалась для изучения компиляции с сохранением типов (League, Trifonov, and Shao, 2001), а также семантических оснований Java (Studer, 2001).

Целью разработки FJ было максимально возможное упрощение и сокращение доказательства типовой безопасности, при сохранении возможности легко распространить его на основные конструкции полной версии Java. Всякая конструкция, которая *удлиняла* доказательство, существенно не *изменяя* его, подлежала удалению. Как и в других исследовательских проектах такого рода, в FJ отсутствуют такие продвинутые возможности языка, как параллелизм и рефлексия. Также отсутствуют и многие другие конструкции Java: присваивание, интерфейсы, перегрузка функций, посылка сообщений переменной `super`, нулевые указатели, базовые типы (`int`, `bool`), объявления абстрактных методов, внутренние классы, замещение полей надкласса полями

подкласса, контроль доступа (`public`, `private` и т. д.), а также исключения. Свойства Java, которые остаются в FJ — взаимно рекурсивные определения классов, создание объектов, доступ к полям, вызов методов, переопределение методов в подклассах, рекурсия между методами с использованием `this`, подтиповование и приведение типов.

Главное упрощение, принятое в FJ — отсутствие присваивания. Мы предполагаем, что значения всех полей объекта устанавливаются в конструкторе и более не меняются. Таким образом, FJ оказывается «функциональным подмножеством» Java, в которой многие идиомы языка, например, использование перечислений, представить невозможно. Тем не менее, это подмножество вычислительно полно (несложно закодировать в нем лямбда-исчисление) и достаточно велико, чтобы включать полезные программы — например, многие программы из книги Феллейсена и Фридмана (Felleisen and Friedman, 1998) написаны в чисто функциональном стиле.

19.2. Обзор

Программа на FJ состоит из набора определений классов и терма, подлежащего вычислению, который соответствует телу метода `main` в Java. Вот несколько типичных определений классов на FJ.

```
class A extends Object { A() { super(); } }

class B extends Object { B() { super(); } }

class Pair extends Object {
    Object fst;
    Object snd;
    // Конструктор:
    Pair(Object fst, Object snd) {
        super(); this.fst=fst; this.snd=snd; }
    // Определение метода:
    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd); } }
```

Ради синтаксического единства мы всегда указываем надкласс (даже если это `Object`), всегда записываем конструктор (даже для тривиальных классов A и B), и всегда указываем объект-получатель при доступе к полю или вызове метода (скажем, `this.snd`), даже если это `this`. Конструкторы всегда имеют одну и ту же жесткую структуру: есть по одному аргументу на каждое поле, с тем же именем, что и у поля; в начале для инициализации полей надкласса вызывается конструктор `super`; затем оставшиеся поля инициализируются значениями соответствующих параметров. В нашем примере надклассом всех приведенных классов служит `Object`, в котором никаких полей нет, так что вызовы `super` не имеют аргументов. Конструкторы — единственное место, в котором в программах на FJ встречаются `super` и оператор `=`. Поскольку в FJ нет операций с побочными эффектами, тело метода всегда состоит из оператора `return`, который возвращает терм, как изображено на примере `setfst()`.

В FJ есть пять видов термов. В нашем примере `new A()`, `new B()` и `new Pair(...)` — *конструкторы объектов* (object constructors), а `<...>.setfst(...)` — *вызов метода* (method invocation). В теле `setfst`, терм `this.snd` является *обращением к полю*, а `newfst` и `this` — переменные.¹ В контексте приведенных определений, терм

```
new Pair(new A(), new B()).setfst(new B())
```

дает при вычислении `new pair(new B(), new B())`.

Последний, пятый вид термов — *приведение типа* (cast) (см. 15.5). Терм

```
((Pair)new Pair(new Pair(new A(), new B()),
               new A()).fst).snd
```

дает при вычислении `new B()`. Подтерм `(Pair)t`, где `t` равняется `new Pair(...).fst`, является приведением типа. Это приведение необходимо, поскольку `t` — обращение к полю `fst`, про которое объявлено, что оно содержит объект типа `Object`, тогда как следующее обращение к полю `snd` определено только для объектов типа `Pair`. Во время выполнения правила вычисления проверят, принадлежит ли объект класса `Object`, хранимый в поле `fst`, классу `Pair` (в данном случае проверка будет успешной).

Отказ от побочных эффектов сам приводит к приятному побочному эффекту: вычисление можно целиком формализовать в рамках синтаксиса FJ, не обращаясь к дополнительным механизмам для моделирования кучи (см. гл. 13). Есть три основных правила вычисления: одно для обращения к полю, одно для вызова метода и одно для приведения типа. Напомним, что в лямбда-исчислении правило вычисления для применений предполагает, что функция сведена к лямбда-абстракции. Аналогично, в FJ правила вычисления предполагают, что объект, над которым ведется работа, сведен к терму `new`. В лямбда-исчислении действует лозунг «все на свете — функции», здесь — «все на свете — объекты».

Следующий пример показывает правило для обращения к полям (E-PROJNEW) в действии:

```
new Pair(new A(), new B()).snd → new B()
```

Поскольку действует жесткий синтаксис конструкторов объектов, то мы знаем, что конструктор содержит по одному параметру для каждого поля, в том же порядке, в котором эти поля объявлены. Здесь это поля `fst` и `snd`, и обращение к полю `snd` выбирает второй параметр.

Вот как действует правило для вызова методов (E-INVKNEW):

```
new Pair (new A(), new B()).setfst(new B())
→ [ newfst ↦ new B(),
    this ↦ new Pair(new A(), new B()) ]
new Pair(newfst, this.snd)
```

т. е., `new Pair(new B(), new Pair(new A(), new B())).snd`

¹ В отличие от Java, в синтаксисе FJ `this` считается переменной, а не ключевым словом

Получателем при вызове метода является объект `new Pair(new A(), new B())`, так что мы ищем метод `setfst` в классе `Pair` и видим, что этот метод имеет формальный параметр `newfst` и тело `new Pair(newfst, this.snd)`. Вызов метода сводится к телу, в котором формальный параметр заменяется на реальный, а особая переменная `this` заменяется на объект-получатель. Это похоже на правило бета-редукции (E-APPABS) в лямбда-исчислении. Основных отличий два: а) класс получателя определяет, где следует искать тело метода (при этом поддерживается переопределение методов в подклассах) и б) переменная `this` заменяется на объект-получатель сообщения (при этом поддерживается «открытая рекурсия через `self`»).² Как и в лямбда-исчислении, в языке FJ ситуация, когда параметр встречается в теле более одного раза, может привести к размножению значений аргумента. Но поскольку не имеется побочных эффектов, это различие со стандартной семантикой Java невозможно обнаружить.

Вот как действует правило для приведения типов (E-CASTNEW):

`(Pair)new Pair(new A(), new B()) → new Pair(new A(), new B())`

После того, как аргумент приведения сведен к объекту, нетрудно проверить, что класс конструктора является подклассом класса, к которому он приводится. Если это так (как в нашем случае), правило редукции убирает приведение. В противном случае, как в терме `(A)new B()`, никакое правило не применимо, и вычисление оказывается в *тупике* (*stuck*), что приводит к сообщению об ошибке времени выполнения.

Вычисление может оказаться в тупике в одном из трех случаев: при попытке обратиться к полю, не объявленному в данном классе; при попытке вызвать метод, не определенный для данного класса («непонятное сообщение»), либо при попытке привести объект к чему-то, что не является надклассом текущего класса объекта. Мы докажем, что первые два случая никогда не встречаются в правильно типизированных программах, а третий никогда не встречается в правильно типизированных программах без приведения вниз (а также без «глупых приведений» — этот технический термин разъясняется ниже).

Мы принимаем стандартную стратегию вычисления с вызовом по значению. Вот шаги вычисления второго приведенного нами терма-примера. Подлежащий редукции подтерм на каждом шаге подчеркнут.

```
((Pair) (new Pair(new Pair(new A(),new B()), new A())).fst.snd)
→ ((Pair)new Pair(new A(),new B())).snd
→ new Pair(new A(),new B()).snd
→ new B()
```

19.3. Именные и структурные системы типов

Прежде чем перейти к формальному определению FJ, следует исследовать одно существенное стилистическое различие между FJ (и Java), с одной стороны, и вариантами типизированного лямбда-исчисления (рассматриваемого

²Читатель, знакомый с исчислением объектов Абади и Карделли (Abadi and Cardelli, 1996), увидит много общего и с их правилом ς -редукции.

в этой книге), с другой. Различие это касается статуса *имен типов* (type names).

В предыдущих главах мы часто вводили краткие имена для длинных или сложных типов, чтобы улучшить читаемость примеров, например:

```
NatPair = {fst:Nat, snd:Nat};
```

Роль этих определений чисто косметическая: имя `NatPair` служит простым сокращением для `{fst:Nat, snd:Nat}`, и два этих выражения взаимозаменимы в любом контексте. Наши формальные описания различных исчислений не упоминают такие сокращения.

Напротив, в Java, как и во многих других широко распространенных языках, определения типов играют намного более важную роль. Каждый составной тип, используемый в программе на Java, имеет имя, и когда мы объявляем тип локальной переменной, поля или параметра метода, мы всегда делаем это, давая ему имя. «Голые» типы вроде `{fst:Nat, snd:Nat}` просто не могут оказаться в этих синтаксических позициях.

Имена типов играют ключевую роль в отношении подтиповования в Java. Каждый раз, когда вводится новое имя (в определении класса или интерфейса), программист должен явно объявить, какие классы и интерфейсы расширяет (или, в случае нового класса и уже существующего интерфейса, «реализует») новый тип. Компилятор проверяет эти объявления и убеждается, что возможности, предоставляемые новым классом или интерфейсом, действительно расширяют возможности, имеющиеся в каждом надклассе и надинтерфейсе. Эта проверка соответствует подтипованию записей в типизированном лямбда-исчислении. Отношение подтиповования определено между *именами* типов, как рефлексивное и транзитивное замыкание отношения непосредственного подтипа. Если одно имя не объявлено как подтип другого, оно им и не является.

Системы типов, подобные системе Java, в которых имена типов существенны, а подтиповование определяется явно, называются *именными* (nominal). В этой книге по большей части встречаются системы типов, в которых имена не играют большой роли, а подтиповование определяется прямо на структурах типов. Такие системы типов называются *структурными* (structural).

У именных систем типов есть как преимущества, так и недостатки в сравнении со структурными. Возможно, самое большое преимущество состоит в том, что имена типов в именных системах полезны не только на стадии проверки типов, но и во время выполнения. В большинстве именных языков каждый объект во время выполнения помечается словом-заголовком, представленное как указатель на структуру времени выполнения, которая содержит имя типа и указывает на его непосредственные надтипы. Такие метки типов полезны в нескольких отношениях, в том числе, для проверки во время выполнения (например, в Java, для теста `instanceOf` и при операции приведения вниз), для печати, для сериализации структур данных в бинарные массивы, предназначенные для хранения в файлах или передачи по сети, а также для рефлексии, позволяющей программе динамически исследовать поля и методы данного ей объекта. Метки типа, присутствующие во время выполнения, могут поддерживаться и в структурных системах (см. Glew, 1999; League, Shao,

and Trifonov, 1999; League, Trifonov, and Shao, 2001, а также списки литературы в этих работах), однако там они представляют собой дополнительный, отдельный механизм; в именных системах метки типа совпадают с типами, проверяемыми на стадии компиляции.

Менее существенное, но тоже приятное, свойство именных систем состоит в том, что они предлагают естественный и интуитивно ясный подход к *рекурсивным типам* (recursive types), — типам, в определении которых упоминается сам тип. (Мы будем подробно обсуждать рекурсивные типы в гл. 20.) Такие типы встречаются при серьезном программировании на каждом шагу, они требуются для описания таких распространенных структур, как деревья и списки, и в именных системах они поддерживаются самым естественным из возможных способов: в теле его собственного объявления сослаться на *List* столь же просто, как и на любой другой тип. Даже *взаимно* рекурсивные типы не требуют никаких специальных усилий. Множество имен типов рассматривается как заданное с самого начала. Поэтому если в определении типа A упоминается тип B, а в определении типа B — тип A, то не возникает вопроса о том, «кто из них определен раньше». Разумеется, рекурсивные типы можно описать и в структурных системах типов. Более того, высокоуровневые языки вроде ML обычно комбинируют рекурсивные типы с другими возможностями языка, поэтому для программиста их использование выглядит так же естественно, как и в именных языках. Однако в исчислениях, предназначенных для более фундаментальных задач, например, для построения доказательств типовой безопасности, механизмы строгой работы с рекурсивными типами могут оказаться достаточно громоздкими, особенно если разрешены взаимно рекурсивные типы. То, что в именных языках мы получаем рекурсивные типы, по существу, «бесплатно» — их несомненное преимущество.

Еще одно преимущество именных систем состоит в том, что проверка того, является ли один тип подтипов другого, оказывается почти тривиальной. Разумеется, компилятор по-прежнему обязан убедиться в том, что объявленные отношения подтиповирования безопасны. Эта работа, по существу, дублирует структурное отношение подтиповирования, но её требуется проделать только один раз для каждого типа — там, где этот тип определен, — а не при каждой проверке на подтиповирование. Поэтому в программах проверки типов для именных систем типов несколько проще добиться хорошей производительности. Впрочем, в более серьезных компиляторах становится труднее определить, влияет ли различие между структурными и именными системами на производительность, поскольку правильно спроектированные программы проверки типов для структурных систем включают методы представления данных, которые сводят большинство проверок на подтиповирование к простому сравнению, — см. с. 250.

Наконец, еще одно часто упоминаемое преимущество явных объявлений отношения подтиповирования состоит в том, что они избегают «случайного включения», когда проверка типов ошибочно одобряет программу, в которой вместо значения одного типа используется значение другого, структурно совместимого с первым. Этот вопрос вызывает больше споров, чем предыдущие, поскольку есть другие, вероятно, более изящные, способы избегать случайного включения: например, с помощью типов данных с одним конструктором

(с. 156) или абстрактных типов данных (гл. 24).

При всех этих преимуществах (особенно меток типа и простого обращения с рекурсивными типами) неудивительно, что именные системы типов более распространены в промышленных языках программирования. С другой стороны, исследовательская литература по языкам программирования посвящена почти исключительно структурным системам типов.

Одна из непосредственных причин этого состоит в том, что (по крайней мере, без учета рекурсивных типов) структурные системы изящнее и проще. В структурной системе выражение, обозначающее тип — это замкнутая сущность; оно содержит в себе все необходимое для понимания его значения. В именной системе работа всегда ведется с учетом некоторого глобального собрания имен типов и связанных с ними определений. Как правило, и определения, и доказательства из-за этого оказываются многословнее.

Более интересная причина состоит в том, что исследовательская литература обычно имеет дело с более сложными конструкциями — в частности, с мощными механизмами абстракции типов (параметрический полиморфизм, абстрактные типы данных, определяемые пользователем операторы над типами, функторы и т. п.), — и с языками вроде ML или Haskell, в которых эти конструкции встречаются. К сожалению, эти конструкции плохо совместимы с именными системами типов. Например, такой тип, как `List(T)`, кажется несводимо составным — где-то в программе будет только одно определение конструктора `List`, и к этому определению требуется обращаться, чтобы понять поведение `List(T)`, поэтому рассматривать `List(T)` как атомарное имя не получится. Некоторые именные языки были расширены такими «обобщенными» конструкциями (Myers, Bank, and Liskov, 1997; Agesen, Freund, and Mitchell, 1997; Bracha, Odersky, Stoutamire, and Wadler, 1998; Cartwright and Steele, 1998; Stroustrup, 1997), но в результате этих расширений системы типов перестают быть чисто именными, и получаются довольно сложные сочетания двух подходов. Поэтому разработчики языков с богатыми системами типов обычно предпочитают структурный подход.

Прояснение отношений между именными и структурными системами типов остается предметом активных исследований.

19.4. Определения

Мы приступаем к формальному определению FJ.

Синтаксис

Синтаксис FJ приведен на рис. 19.1. Метапеременные A, B, C, D и E имеют своими значениями имена классов; f и g — имена полей; m — имена методов; x — имена параметров; s и t — термы; u и v — значения; CL — объявления классов; K — объявления конструкторов; M — объявления методов. Мы предполагаем, что множество переменных включает особую переменную `this`, но что `this` никогда не служит именем аргумента метода. При этом `this` считается неявно связанной в теле каждого объявления метода. Правило вычисления для вызова методов (правило E-INVKNEW на рис. 19.3) подставляет

<i>Синтаксис</i>		<i>Подтипы</i>	$C <: D$
$CL ::=$	<i>объявления классов:</i>		
	$class C extends C \{ \bar{C} f; K \bar{M} \}$		$C <: C$
$K ::=$	<i>объявления конструкторов:</i>	$C <: D \quad D <: E$	$\frac{}{C <: E}$
	$C(\bar{C} \bar{f}) \{super(\bar{f});$ $this.\bar{f}=\bar{f};\}$	$CT(C) = class C extends D \{ \dots \}$	
$M ::=$	<i>объявления методов:</i>		$C <: D$
	$C m(\bar{C} \bar{x}) \{return t;\}$		
$t ::=$	<i>термы:</i>		
	x <i>переменная</i>		
	$t.f$ <i>обращение к полю</i>		
	$t.m(\bar{t})$ <i>вызов метода</i>		
	$new C(\bar{t})$ <i>создание объекта</i>		
	$(C)t$ <i>приведение типа</i>		
$v ::=$	<i>значения:</i>		
	$new C(\bar{v})$ <i>создание объекта</i>		

Рис. 19.1. Облегченная Java (синтаксис и подтипы)

соответствующий объект вместо переменной `this`, а также подставляет значения аргументов вместо имен параметров.

Мы пишем \bar{f} в качестве сокращения f_1, \dots, f_n (аналогично, $\bar{C}, \bar{x}, \bar{t}$ и т. п.), а также \bar{M} в качестве сокращения $M_1 \dots M_n$ (без запятых). Аналогично мы сокращаем и операции над парами последовательностей, а именно: « $\bar{C} f$ » вместо « $C_1 f_1, \dots, C_n f_n$ », где n — длина последовательностей \bar{C} и \bar{f} , « $\bar{C} \bar{f}$ » вместо « $C_1 f_1; \dots; C_n f_n$ » и «`this.f=f;`» вместо «`this.f1=f1; ...; this.fn=fn;`». Предполагается, что в последовательностях объявлений полей, имен параметров и объявлений методов нет повторяющихся имен.

Объявление `class C extends D { \bar{C} \bar{f}; K \bar{M} }` вводит класс с именем C и надклассом D . Новый класс имеет поля \bar{f} с типами \bar{C} , единственный конструктор K и набор методов \bar{M} . Переменные экземпляра, объявленные внутри C , добавляются к тем, что были объявлены в D и его надклассах. Имена новых переменных должны отличаться от старых.³ Напротив, методы C могут либо переопределять методы с тем же именем, уже присутствующие в D , либо добавлять новую функциональность, присущую исключительно C .

Объявление конструктора `C (\bar{D} g, \bar{C} \bar{f}) {super(\bar{g}); this.\bar{f}=\bar{f};}` показы-

³ В Java переменные экземпляра надкласса могут объявляться заново — такое определение скрывает исходную переменную в текущем классе и его подклассах. В FJ мы опускаем это свойство языка.

вает, как следует инициализировать поля экземпляра C . Форма конструктора полностью определяется объявлениями переменных экземпляра C и его надклассов: конструктор *обязан* принимать в точности столько параметров, сколько имеется переменных экземпляра, и его тело *обязано* состоять из вызова конструктора надкласса, который инициализирует поля параметрами \bar{g} , а затем параметры \bar{f} присваиваются полям с теми же именами, объявленным в классе C . (Эти ограничения проверяются в правиле типизации для классов на рис. 19.4.) В полной Java конструктор подкласса должен содержать вызов конструктора надкласса (если конструктор надкласса принимает аргументы); именно поэтому здесь тело конструктора вызывает `super` для инициализации полей экземпляра. Если бы мы не стремились сделать FJ строгим подмножеством Java, мы могли бы отказаться от вызова `super` и заставить каждый конструктор напрямую инициализировать все переменные.

Объявление метода $D \ m(\bar{C} \ \bar{x}) \ \{\text{return } t;\}$ вводит метод с именем m , типом результата D и параметрами \bar{x} типов \bar{C} . Телом метода является единственный оператор `return t`. Переменные \bar{x} связаны внутри t . Кроме того, считается, что внутри t связана особая переменная `this`.

Таблица классов CT — отображение имен классов C на объявления классов CL . Программа представляет собой пару (CT, t) , состоящую из таблицы классов и терма. Чтобы упростить нотацию в последующем изложении, мы всегда будем предполагать *фиксированную* таблицу классов CT .

У каждого класса есть надкласс, объявляемый через `extends`. Отсюда возникает вопрос: какой надкласс имеет класс `Object`? Есть несколько способов решения этого вопроса; простейшее решение, которое мы здесь принимаем — считать `Object` особо выделенным именем класса, определение которого *отсутствует* в таблице классов. Вспомогательная функция, ищащая поля в таблице классов, рассматривает `Object` как особый случай, и возвращает для него пустую последовательность полей, обозначаемую символом \bullet ; предполагается также, что у класса `Object` нет методов.⁴

По таблице классов можно выявить отношение подтиповования между классами. Запись $C <: D$ будет означать, что C является подтиповом D . То есть, отношение подтиповования — это транзитивно-рефлексивное замыкание отношения непосредственного подтиповования, задаваемого выражениями `extends` в CT . Формально оно определяется на рис. 19.1.

Предполагается, что данная нам таблица классов удовлетворяет некоторым минимальным условиям: а) $CT(C) = \text{class } C\dots$ для каждого $C \in \text{dom}(CT)$; б) $\text{Object} \notin \text{dom}(CT)$; в) для каждого имени класса C (кроме `Object`), встречающегося где-либо в CT , выполняется $C \in \text{dom}(CT)$; г) в отношении подтиповования, построенном на основе CT , нет циклов — т. е., отношение $<:$ антисимметрично.

Заметим, что типам, определяемым таблицей классов, разрешается быть рекурсивными в том смысле, что определение A может использовать A как тип своих методов и переменных экземпляра. Разрешена и взаимная рекурсия между определениями классов.

⁴ В полной Java у класса `Object` есть несколько методов. В языке FJ мы их игнорируем.

<p><i>Поиск полей</i></p> $\text{fields}(C) = \bar{C} \bar{f}$ $\text{fields}(\text{Object}) = \bullet$	<p><i>Поиск методов</i></p> $\text{mbody}(m, C) = (\bar{x}, t)$	
$CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; K \bar{M} \}$ $\text{fields}(D) = \bar{D} \bar{g}$	$CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; K \bar{M} \}$ $B m (\bar{B} \bar{x}) \{ \text{return } t; \} \in \bar{M}$ $mbody(m, C) = (\bar{x}, t)$	
$\text{fields}(C) = \bar{D} \bar{g}, \bar{C} \bar{f}$		
<p><i>Поиск типов</i></p> $mtype(m, C) = \bar{C} \rightarrow C$	<p><i>методов</i></p>	$CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; K \bar{M} \}$ $m \text{ не определен в } \bar{M}$ $mbody(m, C) = mbody(m, D)$
$CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; K \bar{M} \}$ $B m (\bar{B} \bar{x}) \{ \text{return } t; \} \in \bar{M}$ $mtype(m, C) = \bar{B} \rightarrow B$		<p><i>Моррекстное переопределение методов</i></p> $override(m, D, \bar{C} \rightarrow C_0)$ <p>из $mtype(m, D) = \bar{D} \rightarrow D_0$, следует, что $\bar{C} = \bar{D}$ и $C_0 = D_0$</p> $override(m, D, \bar{C} \rightarrow C_0)$
$CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; K \bar{M} \}$ $m \text{ не определен в } \bar{M}$ $mtype(m, C) = mtype(m, D)$		

Рис. 19.2. Облегченная Java (вспомогательные определения)

Упражнение 19.4.1 [★]: По аналогии с правилом S-ТОР, в нашем лямбда-исчислении с подтиปами можно было бы ожидать наличия правила, утверждающего, что *Объект* является надтипом любого класса. Почему оно нам здесь не нужно?

Вспомогательные определения

В правилах типизации и вычисления мы будем пользоваться некоторыми вспомогательными определениями, которые представлены на рис. 19.2. Поля класса *C*, обозначаемые как *fields(C)*, представляют собой последовательность $\bar{C} \bar{f}$, связывающую тип каждого поля с его именем для всех полей, объявленных в классе *C* и его надклассах. Тип метода *m* в классе *C*, обозначаемый *mtype(m, C)* — это пара, которая записывается как $\bar{B} \rightarrow B$, состоящая из последовательности типов аргументов \bar{B} и одного типа результата *B*. Аналогично, тело метода *m* в классе *C*, обозначаемое *mbody(m, C)*, есть пара, записываемая как (\bar{x}, t) , состоящая из последовательности параметров *x* и терма *t*. Предикат *override(m, D, C → C₀)* решает, можно ли определить метод *m* с типами аргументов \bar{C} и типом результата *C₀* в подклассе класса *D*. В случае переопределения, если метод с таким же именем определен в надклассе, новый метод обязан иметь тот же самый тип.

Вычисление	$t \rightarrow t'$
$\frac{\text{fields}(C) = \bar{C} \bar{f}}{(\text{new } C(\bar{v})).f_i \rightarrow v_i}$ (E-PROJNEW)	$\frac{t_0 \rightarrow t'_0}{t_0.m(\bar{t}) \rightarrow t'_0.m(\bar{t})}$ (E-INVK-RECV)
$\frac{mbody(m, C) = (\bar{x}, t_0)}{(\text{new } C(\bar{v})).m(\bar{u})}$	$\frac{t_i \rightarrow t'_i}{v_0.m(\bar{v}, t_i, \bar{t}) \rightarrow v_0.m(\bar{v}, t'_i, \bar{t})}$ (E-INVK-ARG)
$\rightarrow [\bar{x} \mapsto \bar{u}, \text{this} \mapsto \text{new } C(\bar{v})]t_0$ (E-INVKNEW)	
$\frac{C <: D}{(D)(\text{new } C(\bar{v})) \rightarrow \text{new } C(\bar{v})}$ (E-CASTNEW)	$\frac{t_i \rightarrow t'_i}{\text{new } C(\bar{v}, t_i, \bar{t}) \rightarrow \text{new } C(v, t'_i, \bar{t})}$ (E-NEW-ARG)
$\frac{t_0 \rightarrow t'_0}{t_0.f \rightarrow t'_0.f}$ (E-FIELD)	$\frac{t_0 \rightarrow t'_0}{(C)t_0 \rightarrow (C)t'_0}$ (E-CAST)

Рис. 19.3. Облегченная Java (вычисление)

Вычисление

Мы используем стандартную операционную семантику с вызовом по значению (рис. 19.3). Три правила вычисления — для обращения к полю, вызова метода и приведения типа, — были рассмотрены в §19.2. Остальные правила формализуют стратегию вызова по значению. Значения, которые могут получиться при нормальном завершении вычислителя — это полностью вычисленные термы создания объектов вида $\text{new } C(\bar{v})$.

Чтобы совершить операцию приведения типа, нужно проверить, является ли настоящий тип объекта, подвергающегося приведению, подтипом типа, указанного в выражении приведения. Если это так, то операция приведения отбрасывается, а сам объект объявляется ее результатом. Это в точности соответствует семантике Java: приведение во время выполнения никак не изменяет объект, — оно либо завершается успешно, либо терпит неудачу и порождает исключение. В FJ, вместо того, чтобы порождать исключение, неудачное приведение просто оказывается тупиковым термом, поскольку никакое правило вычисления к нему не применимо.

Типизация

Правила типизации для термов, объявлений методов и объявлений классов приведены на рис. 19.4. Контекст Γ представляет собой конечное отображение переменных на типы, записываемое как $\bar{x}:\bar{C}$. Утверждения о типизации для термов имеют вид $\Gamma \vdash t : C$, что читается «в контексте Γ терм

Типизация термов	$\boxed{\Gamma \vdash t : C}$	
$\frac{x:C \in \Gamma}{\Gamma \vdash x : C}$	(T-VAR)	$\frac{\Gamma \vdash t_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)t_0 : C}$
$\frac{\Gamma \vdash t_0 : C_0 \quad fields(C_0) = \bar{C} \bar{f}}{\Gamma \vdash t_0.f_i : C_i}$	(T-FIELD)	$\frac{\Gamma \vdash t_0 : D \quad C \triangleleft D \quad D \triangleleft C}{\Gamma \vdash (C)t_0 : C}$ <i>предупреждение о глупости</i>
$\frac{\Gamma \vdash t_0 : C_0 \quad mtype(m, C_0) = \bar{D} \rightarrow C \quad \Gamma \vdash t : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash t_0.m(t) : C}$	(T-INVK)	(T-SCAST)
$\frac{fields(C) = \bar{D} \bar{f}}{\Gamma \vdash t : \bar{C} \quad \bar{C} <: \bar{D}}$	(T-NEW)	М ОК В С
$\frac{\Gamma \vdash t_0 : D \quad D <: C}{\Gamma \vdash (C)t_0 : C}$	(T-UCAST)	$\frac{\bar{x}:\bar{C}, this:C \vdash t_0:E_0 \quad E_0 <: C_0 \quad CT(C) = class\ C\ extends\ D\ \{...\} \quad override(m, D, \bar{C} \rightarrow C_0)}{C_0\ m\ (\bar{C}\ \bar{x})\ \{return\ t_0\}\ OK\ в\ C}$
		Типизация методов
		OK
		Типизация классов
		OK
		$K = C(\bar{D} \bar{g}, \bar{C} \bar{f}) \quad \{super(\bar{g});\ this.\bar{f}=\bar{f}\}$ $fields(D) = \bar{D} \bar{g} \quad \bar{M} \text{ OK в } C$
		class C extends D { $\bar{C} \bar{f}; K \bar{M}$ } OK

Рис. 19.4. Облегченная Java (тиปизация)

t имеет тип C . Правила типизации управляются синтаксисом,⁵ причем для каждого типа терма имеется по одному правилу, кроме приведений, для которых есть три правила (они обсуждаются ниже). Правила типизации для конструкторов и для вызовов методов проверяют, что тип каждого аргумента является подтипом того, который объявлен для соответствующего формального параметра. Мы сокращаем утверждения о типизации для последовательностей очевидным образом, а именно: $\Gamma \vdash \bar{x}:\bar{C}$ служит сокращением для $\Gamma \vdash x_1:C_1, \dots, \Gamma \vdash x_n:C_n$, а $\bar{C} <: \bar{D}$ для $C_1 <: D_1, \dots, C_n <: D_n$.

Небольшой технической инновацией в FJ является понятие «глупого» приведения типа. Есть три правила для приведений: *восходящее приведение* (upcast), в котором тип приводимого объекта является подклассом целевого типа, *нисходящее приведение* (downcast), в котором тип приводимого объекта — надкласс целевого типа, и *глупое приведение* (stupid cast), в котором между двумя типами нет никакого отношения подтиповования. Компилятор Java отвергает терм, содержащий «глупое приведение», как неверный. Одна-

⁵Как и в полной Java, мы выбираем алгоритмическую формулировку для отношения типизации. Интересно заметить, что в настоящей Java только такую формулировку можно выбрать. См. упражнение 19.4.6.

ко, мы вынуждены разрешить «глупые приведения» в FJ, если хотим обеспечить безопасность типов с помощью теоремы о сохранении типов в семантике с малым шагом. Это происходит потому, что разумный терм может при редукции превратиться в терм, содержащий «глупое приведение». Рассмотрим, например, следующее выражение, в котором используются классы A и B, определенные в §19.2:

$$(A)(\underline{\text{Object}})\text{new } B() \rightarrow (A)\text{new } B()$$

Мы указываем на особую природу «глупых приведений», добавляя *предупреждение о глупости* (stupid warning) в правило T-SCAST; типизация в FJ соответствует типизации в Java только в том случае, если это правило не встречается при выводе.

Утверждения о типизации для объявлений методов имеют вид M OK в C, что читается как «объявление метода M сформулировано верно, если оно встречается в классе C». При выводе используется отношение типизации для тела метода, в котором свободными переменными служат аргументы метода с объявленными для них типами, плюс особая переменная `this` типа C.

Утверждения о типизации для объявлений классов имеют вид CL OK, что читается как «объявление класса CL сформулировано верно». При выводе проверяется, что конструктор вызывает `super` с полями надкласса и инициализирует поля, объявленные в самом классе, а также что все объявления методов в классе верны.

Тип терма может зависеть от типа любого метода, который вызывается внутри терма, а тип метода зависит от типа терма (его тела). Поэтому мы должны убедиться, что здесь не возникает порочного круга. В самом деле, порочный круг не может появиться потому, что тип каждого метода объявляется явно. Действительно, можно загрузить таблицу классов целиком и использовать ее для проверки типов, не требуя, чтобы все классы в этой таблице были проверены. Достаточно, чтобы каждый класс был рано или поздно проверен.

Упражнение 19.4.2 [★★]: Некоторые проектные решения в FJ продиктованы желанием сделать язык подмножеством Java в том смысле, чтобы каждая правильно или неправильно типизированная программа на FJ была бы, соответственно, правильно или неправильно типизирована на Java, и поведение правильно типизированных программ должно быть одинаково в обоих языках. Предположим, мы отказываемся от этого требования — т. е. предположим, нам достаточно иметь Java-подобное базовое исчисление. Как можно изменить FJ, чтобы язык стал проще или изящнее?

Упражнение 19.4.3 [РЕКОМЕНДУЕТСЯ, ★★★ →]: Операция присваивания нового значения полю объекта опущена в FJ ради простоты изложения, однако можно ее добавить, не сильно изменяя характер исчисления. Сделайте это, используя в качестве образца подход ко ссылкам из гл. 13.

Упражнение 19.4.4 [★★★ →]: Расширьте FJ аналогами конструкций `try` и `raise` из Java. Используйте в качестве образца описание исключений из гл. 14.

Упражнение 19.4.5 [★★→]: Как и Java, FJ представляет отношение типизации в алгоритмической форме. Правило включения отсутствует; вместо этого, несколько других правил содержат среди своих предпосылок проверку подтипования. Можно ли перестроить систему в более декларативном стиле, исключив все или большинство из этих предпосылок, и добавив вместе этого единое правило включения?

Упражнение 19.4.6 [★★★]: В полной Java наряду с классами имеются интерфейсы (*interfaces*), которые определяют типы методов, но не их реализацию. Интерфейсы полезны, поскольку с их помощью можно построить более богатое отношение подтипования, не ограниченное древовидной формой: у каждого класса есть только один надкласс (из которого он наследует переменные экземпляра и тела методов), но кроме того, он может реализовывать произвольное количество интерфейсов.

Наличие интерфейсов в Java диктует алгоритмический подход к отношению типизации, поскольку именно при таком подходе у каждого типизируемого терма имеется единственный (наименьший) тип. Причина лежит во взаимодействии условных выражений (которые в Java записываются как $t_1 ? t_2 : t_3$) с интерфейсами.

1. Покажите, как можно расширить FJ интерфейсами в стиле Java.
2. Покажите, что при наличии интерфейсов отношение подтипования необязательно замкнуто относительно обединений. (Напомним, что в §16.3 существование обединений играло ключевую роль в доказательстве того, что условные выражения имеют минимальный тип.)
3. Каково правило Java для типизации условных выражений? Разумно ли оно?

Упражнение 19.4.7 [★★★]: В FJ поддерживается ключевое слово *this* из Java, но не поддерживается *super*. Покажите, как его добавить.

19.5. Свойства

Для FJ можно доказать стандартную теорему о сохранении типов.

Теорема 19.5.1 [СОХРАНЕНИЕ]: Если $\Gamma \vdash t : C$ и $t \rightarrow t'$, то $\Gamma \vdash t' : C'$ для некоторого $C' \leq C$.

Доказательство: УПРАЖНЕНИЕ [★★★].

Кроме того, можно доказать один из вариантов стандартной теоремы о продвижении: если программа правильно типизирована, то она может оказаться в тупике только в случае неудачной попытки приведения вниз. В этом случае мы для выявления ошибочного приведения пользуемся механизмом контекстов вычисления.

Лемма 19.5.2 Пусть имеется правильно типизированный терм t .

1. Если $t = \text{new } C_0(\bar{t}).f$, то $\text{fields}(C_0) = \bar{C} \bar{f}$ и $f \in \bar{f}$.
2. Если $t = \text{new } C_0(\bar{t}).m(\bar{s})$, то $mbody(m, C_0) = (\bar{x}, t_0)$ и $|\bar{x}| = |\bar{s}|$.

Доказательство: Не представляет труда.

Определение 19.5.3 Множество контекстов вычисления (*evaluation contexts*) для FJ определяется следующим образом:

E	$::=$	контексты вычисления:
\square		«дыра»
$E.f$		обращение к полю
$E.m(\bar{t})$		вызов метода (получатель)
$v.m(\bar{v}, E, \bar{t})$		вызов метода (аргумент)
$\text{new } C(v, E, \bar{t})$		создание объекта (аргумент)
$(C)E$		приведение типа

Контекст вычисления — это терм с дырой внутри (обозначаемой как \square). Запись $E[t]$ обозначает обычновенный терм, получающийся при замене дыры в E на t .

Контексты вычисления соответствуют понятию «следующего подтерма, подлежащего вычислению», в том смысле, что, имея $t \rightarrow t'$, мы можем выразить t и t' в виде $t = E[r]$ и $t' = E[r']$ для уникальной тройки E , r и r' , так, чтобы выполнялось $r \rightarrow r'$ согласно одному из правил вычисления E-PROJNEW, E-INVKNEW либо E-CASTNEW.

Теорема 19.5.4 [ПРОДВИЖЕНИЕ]: Пусть имеется замкнутый правильно типизированный терм t в нормальной форме. Тогда либо а) t является значением, либо б) для некоторого контекста вычисления E имеем $t = E[(C)(\text{new } D(\bar{v}))]$, причем $D \triangleleft C$.

Доказательство: Прямолинейная индукция по деревьям вывода типов.

Свойство продвижения можно несколько уточнить: если в терме t все приведения направлены вверх, он не может оказаться в тупике (кроме того, если в программе содержатся только приведения вверх, при ее вычислении не могут возникнуть никакие другие приведения). Однако как правило, приведения используются для того, чтобы сузить статический тип объекта, и нам приходится мириться с возможностью того, что во время выполнения приведение может оказаться неудачным. Разумеется, в полной Java неудачное приведение не приводит к немедленной остановке программы: порождается исключение, которое может быть перехвачено активным обработчиком исключений.

Упражнение 19.5.5 [★★★ →]: Используя в качестве заготовки одну из программ проверки типов для какого-либо варианта лямбда-исчисления, постройте программу проверки типов и интерпретатор для Облегченной Java.

Упражнение 19.5.6 [★ ★ ★ →]: В исходной статье по FJ (Igarashi, Pierce, and Wadler, 1999), помимо прочего, было formalизовано понятие полиморфного типа, как в GJ. Расширьте программу проверки типов и интерпретатор из упражнения 19.5.5, включив в него эти конструкции. (Перед выполнением этого упражнения необходимо прочесть гл. 23, 25, 26 и 28.)

19.6. Объекты: кодирование или элементарное понятие?

Мы рассмотрели два различных подхода к семантике и типизации для простых объектно-ориентированных языков. В гл. 18 мы закодировали объекты и наследование реализаций с помощью таких конструкций простого типизированного лямбда-исчисления, как записи, ссылки и подтипы. В этой главе мы изложили прямое описание простого языка, в котором объекты и классы являются элементарными механизмами.

Каждый из этих подходов может быть по-своему полезен. Исследование кодирования объектов обнажает базовые механизмы инкапсуляции и повторного использования и позволяет сравнить их с другими механизмами, предназначенными для аналогичных целей. Кроме того, кодирование помогает понять, как объекты переводятся компиляторами в языки еще более низкого уровня, а также как объекты взаимодействуют с другими возможностями языка. С другой стороны, подход к объектам как к элементарным механизмам позволяет прямо рассуждать об их операционной семантике и поведении с точки зрения типизации; такое представление лучше приспособлено для высокоуровневого проектирования языков и написания документации.

Разумеется, в конечном счете хотелось бы сохранить обе точки зрения — иметь высокоуровневый язык, включающий элементарные конструкции с классами, объектами и т. д., с собственными правилами типизации и операционной семантикой, а также правила перевода этого языка в некоторый язык более низкого уровня, в котором есть только записи и функции (или даже язык еще более низкого уровня, в котором нет ничего, кроме регистров, указателей и последовательностей команд), и, наконец, доказательство того, что перевод на самом деле является корректной реализацией высокоуровневого языка, в том смысле, что перевод сохраняет свойства вычисления и типизации высокоуровневого языка. Существует много реализаций этой исследовательской программы — для самой FJ это проделали Лиг, Трифонов и Шао (League, Trifonov, and Shao, 2001), а для других объектно-ориентированных базовых исчислений — Хоффман и Пирс (Hofmann and Pierce, 1995b), Брюс (Bruce, 1994, 2002), Абади, Карделли и Вишванатан (Abadi, Cardelli, and Viswanathan, 1996), а также другие авторы.

19.7. Дополнительные замечания

Эта глава написана на базе статьи Игараси, Пирса и Уодлера, где впервые упоминался язык FJ (Igarashi, Pierce, and Wadler, 1999). Основное отличие

в представлении материала состоит в том, что здесь, как и вообще на протяжении этой книги, мы использовали операционную семантику с вызовом по значению, тогда как в исходной статье использовалось недетерминистское отношение бета-редукции.

Существует несколько различных доказательств типовой безопасности для подмножеств Java. В самом раннем из них Дроскопулу, Айзенбах и Хуршид (*Drossopoulou, Eisenbach, and Khurshid, 1999*), используя метод, впоследствии механически проверенный Саймом (*Syme, 1997*), доказывают безопасность для существенного подмножества диалекта Java без параллелизма. Подобно FJ, они работают в рамках операционной семантики с малым шагом, но избегают тонкостей «глупого приведения», целиком убирая из языка приведения. Нипков и Охаймб (*Nipkow and von Oheimb, 1998*) приводят механически проверенное доказательство безопасности для несколько более широкого базового языка. Их язык включает в себя приведения, но сформулирован средствами операционной семантики «с большим шагом», и проблема с «глупыми приведениями» в таком контексте не возникает. Флэтт, Кришнамурти и Феллейсен (*Flatt, Krishnamurthi, and Felleisen, 1998a,b*) используют операционную семантику с малым шагом. Они формализуют подмножество, включающее присваивание и приведения. «Глупые приведения» рассматриваются так же, как в FJ. Их система несколько больше FJ по размеру (правила синтаксиса, типизации и операционной семантики примерно втрое больше по объему), а доказательство безопасности, будучи значительно длиннее, имеет примерно такую же сложность, как для FJ.

Из этих трех работ, работа Флэтта, Кришнамурти и Феллейсена ближе всего к FJ в одном важном аспекте: эти авторы, как и мы, ставят своей целью выбор как можно *меньшего* базового исчисления, отражающего только те свойства Java, которые важны для некоторой конкретной задачи. В их случае, задачей был анализ расширения Java *классами-примесями* (*mixins*) в стиле Common Lisp. Напротив, целью двух других упомянутых систем было исследовать как можно *большее* подмножество Java, поскольку основной их интерес заключался в доказательстве безопасности Java как таковой.

В литературе по основам объектно-ориентированных языков можно найти множество статей по формализации объектно-ориентированных языков, основанных на классах, либо рассматривающих классы как элементарный механизм (напр., *Wand, 1989a, Bruce, 1994, Bono, Patel, Shmatikov, and Mitchell, 1999b, Bono, Patel, and Shmatikov, 1999a*), либо переводящих классы в более низкоуровневые конструкции (напр., *Fisher and Mitchell, 1998, Bono and Fisher, 1998, Abadi and Cardelli, 1996, Pierce and Turner, 1994*).

Родственное направление исследований рассматривает модели объектно-ориентированных языков, в которых классы заменяются какой-либо формой *переопределения методов* (*method override*) или *делегации* (*delegation*) (*Ungar and Smith, 1987*), при которых отдельные объекты могут наследовать поведение других *объектов*. Получающиеся исчисления часто бывают проще, чем для языков, основанных на классах, поскольку работают с меньшими наборами базовых понятий. Самое развитое и самое известное из них — *исчисление объектов* (*object calculus*) Абади и Карделли (*Abadi and Cardelli, 1996*). Еще одно популярное исчисление разработано Фишером, Хонселлом и Митчеллом

(Fisher, Honsell, and Mitchell, 1994).

Существенно отличающийся подход к объектам, классам и подтипам, известный под названием *мультиметоды* (multi-methods), был формализован Кастаньей, Гелли и Лонго (Castagna, Ghelli, and Longo, 1995). Дополнительные ссылки можно найти в примечании на с. 254.

В каждом большом языке
заключен маленький, который
стремится выбраться наружу.

Игараси, Пирс и Уодлер
(Igarashi, Pierce, and Wadler,
1999)

В каждой большой программе
заключена маленькая, которая
стремится выбраться наружу.

Тони Хоар, «Эффективное
производство больших
программ» (1970)

Я толстый, но внутри я худой.
Вам никогда не приходило в
голову, что у каждого толстяка
внутри сидит худой человек?

Джордж Оруэлл, «За глотком
свежего воздуха» (1939)

Часть IV

Рекурсивные типы

Глава 20

Рекурсивные типы

В разделе 11.12 мы видели, как можно расширить простую систему типов конструктором типов `List(T)`, который описывает списки с элементами типа `T`. Списки — лишь один пример большого класса распространенных структур, в который также входят очереди, бинарные деревья, помеченные деревья, деревья абстрактного синтаксиса и т. п., которые могут расти до произвольного размера, но при этом имеют простую, регулярную структуру. Например, элемент типа `List(Nat)` всегда является либо пустым (`nil`), либо парой («ячейкой `cons`»), состоящей из числа и `List(Nat)`. Очевидно, не имеет смысла представлять все такие структуры в виде отдельных элементарных языковых конструкций. Вместо этого нам нужен общий механизм, с помощью которого их в случае необходимости можно определить на основе более простых элементов. Этот механизм называется *рекурсивными типами* (recursive types).

Рассмотрим еще раз тип списков чисел¹. Утверждение, что список является либо `nil`, либо парой, мы можем представить при помощи вариантовых типов и типов кортежей, которые определены в разделах 11.10 и 11.7:

```
NatList = <nil:Unit, cons:{..., ...}>;
```

Значение, связанное с меткой варианта `nil`, тривиально, поскольку уже сама метка `nil` сообщает нам все, что требуется знать о пустом списке. В то же время, значение, связанное с меткой `cons`, является парой, состоящей из числа и другого списка. Первый компонент этой пары имеет тип `Nat`,

```
NatList = <nil:Unit, cons: {Nat, ...}>;
```

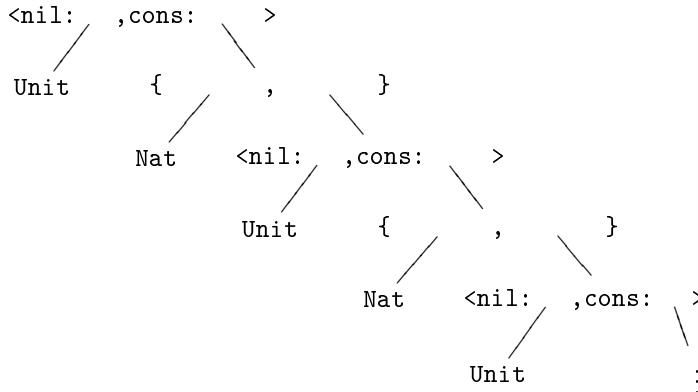
а второй компонент является списком чисел, т. е. элементом того самого типа `NatList`, который мы в данный момент определяем:

```
NatList = <nil:Unit, cons:{Nat,NatList}>;
```

В этой главе рассматривается простое типизированное исчисление с рекурсивными типами. В примерах используются различные конструкции из предыдущих глав; соответствующий интерпретатор называется `fullequirec`. Для раздела 20.2 интерпретатором служит `fullisorec`.

¹ В этой главе мы не будем касаться вопроса о том, как дать единое обобщенное определение списков с элементами произвольного типа `T`. Для этого определения нам понадобится механизм *операторов над типами* (type operators), который будет введен в главе 29.

Это уравнение не является обычным определением в том смысле, что в нем мы не даем новое имя выражению, значение которого нам заранее понятно, — поскольку в правой части используется то самое имя, которое подлежит определению. Напротив, это уравнение можно рассматривать как определение бесконечного дерева:



Рекурсивное уравнение, определяющее это бесконечное дерево, похоже на уравнение, которым определяется рекурсивная функция факториала, приведенное на с. 72. Как и там, нам будет удобно преобразовать это уравнение в правильное определение, переместив «цикл» в правую часть выражения.² Мы сделаем это, вводя явный оператор рекурсии для типов — оператор μ :

```
NatList =  $\mu X$ . <nil:Unit, cons:{Nat,X}>;
```

Интуитивно это выражение читается так: «Пусть NatList будет бесконечным типом, удовлетворяющим уравнению $X = \langle\text{nil:Unit, cons:Nat,X}\rangle$ ».

В разделе 20.2 мы увидим, что существуют два немного разных способа формализации рекурсивных типов — так называемые *эквирекурсивное* (equi-recursive) и *изорекурсивное* (iso-recursive) представления. Они отличаются количеством аннотаций типов, которые программист должен выписать, чтобы помочь системе проверки. В примерах, приведенных в следующем разделе, мы будем использовать более простое эквирекурсивное представление.

20.1. Примеры

Списки

Прежде всего, закончим пример со списками чисел, начатый нами в предыдущем разделе. Чтобы использовать эти списки при программировании, нам нужна константа `nil`, конструктор `cons` для добавления элементов в начало

² Такое перемещение удобно потому, что оно позволяет нам говорить о рекурсивных типах, не именуя их. Однако, работа с явно именованными рекурсивными типами также имеет свои преимущества — см. обсуждение *именных* (nominal) и *структурных* (structural) систем типов в §19.3.

существующего списка, операция `isnil`, которая принимает в качестве аргумента список и возвращает булево значение, а также операции декомпозиции `hd` и `tl` для извлечения, соответственно, первого элемента и остатка из непустого списка. На рис. 11.13 мы определили эти функции как встроенные операторы; теперь наша задача заключается в том, чтобы построить их из базовых элементов.

Определения `nil` и `cons` немедленно следуют из определения `NatList` через двухэлементный вариантный тип.

```
nil = <nil=unit> as NatList;

▷ nil : NatList

cons = λn:Nat. λl:NatList. <cons={n,l}> as NatList;

▷ cons : Nat → NatList → NatList
```

(Напомним из §11.12, что выражение вида `<l=t> as T` используется для создания значений вариантного типа: значение `t` помечается меткой `l` и «вкладывается» в вариантный тип `T`. Заметим также, что процедура проверки типов автоматически «развернет» рекурсивный тип `NatList` в вариантный тип `<nil:Unit, cons:Nat,NatList>`.)

Остальные базовые операции над списками проверяют их структуру и извлекают отдельные их части. Все они реализуются в терминах `case`.

```
isnil = λl:NatList. case l of
    <nil=u> ⇒ true
    | <cons=p> ⇒ false;

▷ isnil : NatList → Bool

hd = λl:NatList. case l of <nil=u> ⇒ 0 | <cons=p> ⇒ p.1;

▷ hd : NatList → Nat

tl = λl:NatList. case l of <nil=u> ⇒ l | <cons=p> ⇒ p.2;

▷ tl : NatList → NatList
```

Для определенности мы решили, что вызов `hd` для пустого списка будет возвращать 0, а вызов `tl` для пустого списка — пустой список. В этих случаях также можно было бы возбуждать исключения.

С помощью всех этих определений можно написать рекурсивную функцию, которая суммирует элементы списка:

```
sumlist = fix (λs:NatList → Nat. λl:NatList.
    if isnil l then 0 else plus (hd l) (s (tl l)));

▷ sumlist : NatList → Nat
mylist = cons 2 (cons 3 (cons 5 nil));
sumlist mylist;

▷ 10 : Nat
```

Заметим, что хотя сам `NatList` — бесконечное типовое выражение, все его элементы являются *конечными* списками. Это потому, что ни примитивы построения пары и пометки вариантов, ни конструкция `fix` с вызовом по значению не пригодны для построения бесконечно больших структур.

Упражнение 20.1.1 [★★]: Один из способов представления помеченных двоичных деревьев состоит в том, что дерево может быть либо листом (без метки), либо внутренним узлом с численной меткой и двумя дочерними деревьями. Определите тип `NatTree` и соответствующий набор операций для создания, уничтожения и проверки деревьев. Напишите функцию, которая будет выполнять обход дерева «в глубину» и возвращать список найденных меток. Для тестирования кода используйте интерпретатор `fullquirec`.

«Жадные» функции

Другой пример, демонстрирующий несколько более хитрое использование рекурсивных типов — тип «жадных функций», которые могут принимать любое количество численных аргументов, и всегда возвращают новую функцию, столь же «жадную»:

```
Hungry = μA. Nat → A;
```

Элементы этого типа могут быть определены с использованием оператора `fix`:

```
f = fix (λf: Nat → Hungry. λn:Nat. f);
```

```
▷ f : Hungry
```

```
f 0 1 2 3 4 5;
```

```
▷ <fun> : Hungry
```

Потоки

Более полезный вариант вышеописанного типа `Hungry` — тип `Stream` (поток), состоящий из функций, которые могут потреблять любое количество значений `unit`, каждый раз возвращая пару, состоящую из числа и нового потока.

```
Stream = μA. Unit → {Nat, A};
```

Мы можем определить две операции декомпозиции для потоков; если `s` — поток, то результатом вызова `hd s` будет первое число, которое он вернет, когда мы передадим ему `unit`.

```
hd = λs:Stream. (s unit).1;
▷ hd : Stream → Nat
```

Аналогично, результатом `tl s` будет новый поток, который мы получим, если передадим `unit` в `s`.

```
tl = λs:Stream. (s unit).2;
▷ tl : Stream → Stream
```

Для создания потока мы снова используем оператор `fix`:

```
upfrom0 = fix (λf: Nat → Stream. λn:Nat. λ_:Unit. {n,f (succ n)}) 0;
▷ upfrom0 : Stream

hd upfrom0;
▷ 0 : Nat

hd (tl (tl (tl upfrom0)));
▷ 3 : Nat
```

Упражнение 20.1.2 [РЕКОМЕНДУЕТСЯ, ★★]: *Создайте поток, выдающий элементы последовательности чисел Фибоначчи (1, 1, 2, 3, 5, 8, 13, ...).*

Потоки можно дальше обобщить до простой формы *процессов* (processes) — функций, принимающих число, а возвращающих число и новый процесс.

```
Process = μA. Nat → {Nat, A};
```

Вот, например, процесс, который на каждом шаге возвращает сумму всех элементов, полученных им до сих пор:

```
p = fix (λf: Nat → Process. λacc:Nat. λn:Nat.
           let newacc = plus acc n in
           {newacc, f newacc}) 0;
▷ p : Process
```

Как и для потоков, можно определить вспомогательные функции для взаимодействия с процессами:

```
curr = λs:Process. (s 0).1;
▷ curr : Process → Nat

send = λn:Nat. λs:Process. (s n).2;
▷ send : Nat → Process → Process
```

Если мы передадим процессу `p` числа 5, 3 и 20, то число, которое он вернет в ответ на последнее обращение, будет 28.

```
curr (send 20 (send 3 (send 5 p)));
▷ 28 : Nat
```

Объекты

Небольшое переосмысление последнего примера дает нам еще один знакомый язык взаимодействия с данными — **объекты**. Вот, например, тип для объектов-счетчиков (**counter**), которые хранят внутри себя число и позволяют либо запрашивать текущее значение этого числа, либо увеличивать его:

```
Counter = μC. {get:Nat, inc:Unit → C};
```

Заметим, что представление объектов здесь — чисто функциональное (как и в главе 19, и в отличие от того, которое было в главе 18): отправка сообщения `inc` объекту-счетчику не вызывает изменения внутреннего состояния объекта; вместо этого операция возвращает новый объект-счетчик, в котором внутреннее состояние увеличено. В этом случае использование рекурсивных типов позволяет указать, что возвращаемый объект имеет тот же самый тип, что и оригинал.

Единственное отличие между этими объектами и процессами, которые мы обсуждали выше, состоит в том, что объект является рекурсивно определенной *записью* (содержащей функцию), тогда как процесс — рекурсивно определенная *функция* (возвращающая кортеж). Польза от такой смены точки зрения в том, что мы можем расширить нашу запись и включить в нее более одной функции, например, добавив операцию уменьшения значения:

```
Counter = μC. {get:Nat, inc:Unit → C, dec:Unit → C};
```

Для создания объекта `counter` мы используем комбинатор неподвижной точки, как мы это делали выше.

```
c = let create = fix (λf: {x:Nat} → Counter. λs: {x:Nat}.
    {get = s.x,
     inc = λ_:Unit. f {x=succ(s.x)},
     dec = λ_:Unit. f {x=pred(s.x)} })
in create {x=0};
```

```
▷ c : Counter
```

При вызове операций над `c` мы просто выделяем соответствующее поле:

```
c1 = c.inc unit;
c2 = c1.inc unit;
c2.get;
```

```
▷ 2 : Nat
```

Упражнение 20.1.3 [★★]: Расширьте тип `Counter` и счетчик `c`, приведенный выше, так, чтобы они включали операции `backup` и `reset` (как мы это делали в §18.7): при выполнении операции `backup` счетчик сохраняет свое текущее значение в специальном внутреннем регистре, а вызов `reset` приводит к восстановлению значения из этого регистра.

Рекурсивные значения на основе рекурсивных типов

Более удивительное использование рекурсивных типов, ясно показывающее их выразительную силу — это построение правильно типизированного комбинатора неподвижной точки. Для любого типа `T` мы следующим образом можем определить конструктор неподвижной точки для функции над `T`.

```
fixT = λf:T → T. (λx:(μA. A → T). f (x x)) (λx:(μA. A → T). f (x x));
```

```
▷ fixT : (T → T) → T
```

Заметим, что если мы сотрем типы, то этот терм будет выглядеть в точности как бестиповый комбинатор неподвижной точки, рассмотренный нами на с. 85.

Здесь основной трюк заключается в использовании рекурсивного типа для типизации двух упоминаний выражения x . Как мы видели в упражнении 9.3.2, типизация этого терма требует, чтобы x имел функциональный тип, областью значений которого является сам x . Ясно, что конечного типа с таким свойством не существует, но бесконечный тип $\mu A. A \rightarrow T$ прекрасно справляется с этой задачей.

Из этого примера следует, что присутствие рекурсивных типов ломает свойство строгой нормализации: с помощью комбинатора fix_T можно написать правильно типизированный терм, вычисление которого (при применении к unit) никогда не завершится.

```
diverget = λ_:Unit. fixT (λx:T. x);  
⇒ diverget : Unit → T
```

Более того, поскольку такие термы можно построить для любого типа, отсюда следует, что каждый тип в этой системе содержит какие-нибудь термы, в отличие от λ_\rightarrow .³

Интерпретация бестипового лямбда-исчисления

Возможно, лучше всего силу рекурсивных типов иллюстрирует тот факт, что бестиповое лямбда-исчисление можно полностью включить, — правильно типизированным образом, — в статически типизированный язык с рекурсивными типами. Допустим, что D — это следующий тип.⁴

```
D = μX. X → X;
```

Определим «функцию вложения» lam , отображающую функции из D в D на элементы D :

```
lam = λf:D → D. f as D;  
⇒ lam : D
```

Для применения одного элемента D к другому мы просто раскрываем тип первого, выделяем функцию и применяем ее ко второму:

```
ap = λf:D. λa:D. f a;  
⇒ ap : D
```

³Этот факт делает системы с рекурсивными типами бесполезными в качестве логик, если мы рассматриваем типы как логические утверждения согласно соотношению Карри-Говарда (см. §9.4), и интерпретируем «тип T содержит термы» как «утверждение T является доказуемым». При этом если каждый тип содержит термы, то значит, что каждое утверждение в логике доказуемо — то есть, что логика противоречива.

⁴Читатель, знакомый с денотационной семантикой, заметит, что определение D в точности совпадает с определяющим свойством *универсальных доменов* (universal domains), используемых в семантических моделях чистого лямбда-исчисления.

Предположим теперь, что M — закрытый лямбда-терм, содержащий только переменные, абстракции и применения. В таком случае, мы можем сконструировать элемент D , представляющий M (обозначив его как M^*), единообразным методом:

$$\begin{aligned} x^* &= x \\ (\lambda x. M)^* &= \text{lam } (\lambda x: D. M^*) \\ (M N)^* &= \text{ap } M^* N^* \end{aligned}$$

Вот, например, бестиповой комбинатор неподвижной точки, выраженный как элемент D :

```
fixD = lam (λf:D.
    ap (lam (λx:D. ap f (ap x x)))
    (lam (λx:D. ap f (ap x x))));
```

▷ fixD : D

Это внедрение чистого лямбда-исчисления можно расширить и включить в него другие элементы, например, числа. Заменим определение D на вариантный тип с одним вариантом для чисел и одним для функций:

```
D = μX. <nat:Nat, fn:X → X>;
```

Таким образом, элемент D является либо числом, либо функцией из D в D , соответственно помеченной `nat` или `fn`. Реализация конструктора `lam` практически такая же, как и выше:

```
lam = λf:D → D. <fn=f> as D;
```

▷ lam : (D → D) → D

Однако реализация `ap` имеет интересное отличие:

```
ap = λf:D. λa:D.
  case f of
    <nat=n> ⇒ diverge_D unit
    | <fn=f> ⇒ f a;
▷ ap : D → D → D
```

Прежде чем нам удастся применить `f` к `a`, нужно выделить функцию `f` с помощью `case`. При этом требуется указать, как наше применение ведет себя, если `f` не является функцией. (В этом примере мы просто зацикливаемся; можно было бы также возбудить исключение). Заметьте, насколько проверка метки в этом примере похожа на проверку метки во время выполнения в динамически типизированных языках вроде Scheme. В этом смысле можно сказать, что типовое вычисление «включает в себя» бестиповое или динамически типизированное вычисление.

Аналогичная проверка меток необходима при определении функции следования для элементов D :

```
suc = λf:D. case f of
  <nat=n> ⇒ (<nat=succ n> as D)
  | <fn=f> ⇒ diverge_D unit;
```

▷ suc : D → D

Включение 0 в D тривиально:

```
zro = <nat=0> as D;
▷ zro : D
```

Упражнение 20.1.4 [★]: Дополните это кодированием булевыми значениями и условными выражениями, и закодируйте термы, *if false then 1 else 0* и *if false then 1 else false* в качестве элементов D. Что происходит при вычислении этих термов?

Упражнение 20.1.5 [РЕКОМЕНДУЕТСЯ, ★★]: Расширьте тип данных D так, чтобы он включал записи

```
D = μX. <nat:Nat, fn:X → X, rcd:Nat → X>;
```

и реализуйте создание записей и проекцию полей. Для простоты в качестве меток полей используйте натуральные числа, — т.е. записи представляются как функции из натуральных чисел в элементы D. Проверьте свое расширение интерпретатором *fullequirec*.

20.2. Формальные определения

В литературе по системам типов представлены два основных подхода к рекурсивным типам. Основное отличие между ними заключается в ответе на простой вопрос: каково отношение между типом $\mu X.T$ и его одношаговой разверткой? Например, как соотносятся NatList и $\langle \text{nil:Unit}, \text{cons:Nat,NatList} \rangle$?

- При *эквирекурсивном* (equi-recursive) подходе эти два типовых выражения рассматриваются как *эквивалентные по определению* (definitionally equal) — взаимозаменяемые во всех контекстах, — поскольку они обозначают одно и то же бесконечное дерево.⁵ Процедура проверки типов отвечает за то, чтобы термы одного типа были допустимы в качестве аргументов функции, ожидающей термы другого, и т. п.

Приятной особенностью эквирекурсивного подхода является то, что они отличаются от декларативных представлений уже известных нам систем только тем, что требуют разрешить, чтобы типовые выражения были бесконечными.⁶ Существующие определения, теоремы о типовой безопасности и доказательства остаются без изменений, если они не зависят от индукции по типовым выражениям (которая, естественно, больше не работает).

Конечно, *реализация* эквирекурсивных типов требует некоторой работы, поскольку алгоритмы проверки типов не могут напрямую работать с бесконечными структурами. Детали таких реализаций описаны в главе 21.

⁵ Отображение из μ -типов в бесконечные деревья формально определяется в §21.8.

⁶ Строго говоря, следует сказать *регулярными* (regular) — см. §21.7.

2. С другой стороны, при *изорекурсивном* (iso-recursive) подходе рекурсивный тип и результат шага его развертки считаются различными, но *изоморфными* типами.

С формальной точки зрения, раскрытием рекурсивного типа $\mu X.T$ является тип, полученный путем взятия тела T и замены всех вхождений X на весь рекурсивный тип, т. е., используя стандартную нотацию для подстановки, $[X \mapsto (\mu X.T)]T$. Например, тип *NatList*, т. е.

$$\mu X.\langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, X\} \rangle,$$

раскрывается в

$$\langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, \mu X.\langle \text{nil} : \text{Unit}, \text{cons} : \{\text{Nat}, X\} \rangle \} \rangle.$$

В системе с изорекурсивными типами для каждого рекурсивного типа $\mu X.T$ мы вводим пару функций

$$\begin{aligned}\text{unfold}[\mu X.T] &: \mu X.T \rightarrow [X \mapsto \mu X.T]T \\ \text{fold}[\mu X.T] &: [X \mapsto \mu X.T]T \rightarrow \mu X.T\end{aligned}$$

которые «свидетельствуют об изоморфизме» через отображение (в обе стороны) значений между двумя типами:

$$\begin{array}{ccc} & \text{unfold}[\mu X.T] & \\ \mu X.T & \xrightarrow{\hspace{2cm}} & [X \rightarrow \mu X.T] T \\ & \xleftarrow{\hspace{2cm}} & \text{fold}[\mu X.T] \end{array}$$

Отображения *fold* и *unfold* реализованы в качестве примитивов языка, как это показано на рис. 20.1. То, что они образуют изоморфизм, отражается в правиле вычисления E-UNFLDFLD, уничтожающем выражение *fold*, когда оно оказывается в паре с соответствующим *unfold*. (Правило вычисления не требует, чтобы аннотации типов для *fold* и *unfold* были одинаковыми, поскольку для проверки такого ограничения нам пришлось бы запускать процедуру проверки типов во время выполнения. Однако, при вычислении правильно типизированной программы эти две аннотации типов *будут равны* при всяком применении E-UNFLDFLD.)

Оба подхода широко используются как в теоретических исследованиях, так и при проектировании языков программирования. Эквирекурсивный стиль, пожалуй, легче понять интуитивно, но он более требователен к процедуре проверки типов, поскольку она должна эффективно находить точки, в которых следует добавить аннотации *fold* и *unfold*. Более того, взаимодействие между эквирекурсивными типами и другими нетривиальными типовыми возможностями, такими, как ограниченная квантификация и операторы над типами, бывает достаточно сложным и приводит к значительным теоретическим трудностям (см., например, Ghelli, 1993; Colazzo and Ghelli, 1999) или даже к тому, что задача проверки типов оказывается неразрешимой (Solomon, 1978).

Изорекурсивный стиль несколько тяжелее по части обозначений — он требует, чтобы программы сопровождались инструкциями *fold* и *unfold* везде,

$\rightarrow \mu$ Расширяем λ_\rightarrow (9.1)

$t ::= \dots$	<i>термы</i>
fold [T] t	<i>свертка</i>
unfold [T] t	<i>развертка</i>

$v ::= \dots$	<i>значения</i>
fold [T] v	<i>свертка</i>

$T ::= \dots$	<i>типы</i>
X	<i>типовая переменная</i>
$\mu X.T$	<i>рекурсивный тип</i>

Новые правила вычисления

 $t \rightarrow t'$

$$\text{unfold } [S] (\text{fold } [T] v_1) \rightarrow v_1 \quad (\text{E-UNFLDFLD})$$

$$\frac{t_1 \rightarrow t'_1}{\text{fold } [T] t_1 \rightarrow \text{fold } [T] t'_1} \quad (\text{E-FLD})$$

$$\frac{t_1 \rightarrow t'_1}{\text{unfold } [T] t_1 \rightarrow \text{unfold } [T] t'_1} \quad (\text{E-UNFLD})$$

Новые правила типизации $\Gamma \vdash t : T$

$$\frac{U = \mu X.T_1 \quad \Gamma \vdash t_1 : [X \mapsto U]T_1}{\Gamma \vdash \text{fold } [U] t_1 : U} \quad (\text{T-FLD})$$

$$\frac{U = \mu X.T_1 \quad \Gamma \vdash t_1 : U}{\Gamma \vdash \text{unfold } [U] t_1 : [X \mapsto U]T_1} \quad (\text{T-UNFLD})$$

Рис. 20.1. Изорекурсивные типы ($\lambda\mu$)

где употребляются рекурсивные типы. Однако на практике эти аннотации часто можно «спрятать», слив их с аннотациями другого рода. Например, в языках семейства ML каждое определение `datatype` неявно вводит рекурсивный тип. Каждое использование конструктора при построении значения конкретного типа данных неявно включает `fold`, а конструктор, используемый при сопоставлении с образцом, неявно применяет `unfold`. Аналогично, в Java каждое определение класса вводит рекурсивный тип, а выполнение метода объекта включает неявный `unfold`. Это удачное перекрытие механизмов делает изорекурсивный стиль достаточно удобным для практического использования.

Вот пример `NatList` в изорекурсивном виде. Удобно будет сначала определить сокращение для развернутой формы `NatList`:

```
NLBody = <nil:Unit, cons:{Nat,NatList}>;
```

Теперь можно определить `nil`, построив вариант с типом `NLBody`, а затем свернув его до `NatList`; `cons` строится аналогичным образом.

```
nil = fold [NatList] (<nil=unit> as NLBody);
cons = λn:Nat. λl:NatList. fold [NatList] <cons={n,l}> as NLBody;
```

Соответственно, определения операций `isnil`, `hd` и `tl` требуют взять `NatList` и рассмотреть его как вариантный тип, чтобы обработать каждый вариант по отдельности. Это делается путем развертки аргумента 1:

```
isnil = λl:NatList.
    case unfold [NatList] l of
        <nil=u> => true
    | <cons=p> => false;
hd = λl:NatList.
    case unfold [NatList] l of
        <nil=u> => 0
    | <cons=p> => p.1;
tl = λl:NatList.
    case unfold [NatList] l of
        <nil=u> => l
    | <cons=p> => p.2;
```

Упражнение 20.2.1 [РЕКОМЕНДУЕТСЯ, ★★]: *Переформулируйте некоторые примеры из §20.1 (в частности, пример `fix` с с. 302) с явным использованием аннотаций `fold` и `unfold`. Проверьте их с помощью интерпретатора `fullisorec`.*

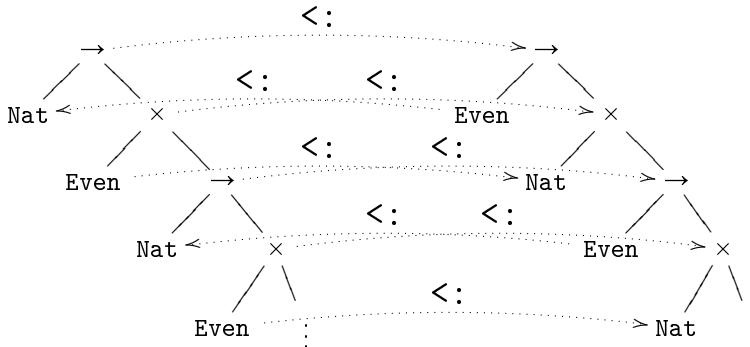
Упражнение 20.2.2 [★★]: *Сделайте набросок доказательства теорем о продвижении и сохранении для изорекурсивной системы.*

20.3. Подтипы

Последний вопрос, который следует обсудить в этой главе, касается сочетания рекурсивных типов с другим, уже рассмотренным, важным расширением простого типизированного лямбда-исчисления — подтиповами. Например, предположим, что тип `Even` является подтипом `Nat`. Как должны соотноситься типы $\mu X. Nat \rightarrow (Even \times X)$ и $\mu X. Even \rightarrow (Nat \times X)$?

Проще всего рассматривать такие вопросы, изучая их «в пределе», — т.е. используя эквириекурсивное толкование рекурсивных типов. В представленном примере элементы обоих типов могут рассматриваться как простые реактивные процессы (ср. с. 301): получив число, они возвращают другое число и новый процесс, готовый получить число, и т.д. Процессы, относящиеся к первому типу, всегда выдают четные числа, а в качестве аргументов могут принимать любые числа. Процессы, относящиеся ко второму типу, могут выдавать произвольные числа, но всегда ожидают, что им передадут четное число. Для первого типа действуют более жесткие ограничения на то, какие аргументы могут быть переданы функции, и какие результаты могут быть

возвращены; поэтому мы интуитивно предполагаем, что первый тип является подтиповм второго. Эти вычисления можно изобразить следующим образом:



Можно ли сделать это интуитивное рассуждение точным? Да, это возможно, как мы увидим в главе 21.

20.4. Дополнительные замечания

Применение рекурсивных типов в информатике восходит как минимум к Моррису (Morris, 1968). Их основные синтаксические и семантические свойства (без подтипов) собраны у Кардоне и Коппо (Cardone and Coppo, 1991). Обзор свойств бесконечных и регулярных деревьев приведен у Курселя (Courcelle, 1983). Основные синтаксические и семантические свойства рекурсивных типов без потипов были установлены в ранних публикациях Юэ (Huet, 1976), а также Маккуина, Плоткина и Сети (MacQueen, Plotkin, and Sethi, 1986). (Еще более ранняя статья Клода Пера (Claude Pair), которая называется «О синтаксисе Алгола 68» («Concerning the Syntax of Algol 68»), содержит, по-видимому, первое доказательство разрешимости равенства для эквирикурсивных типов. Оригинал, опубликованный в журнале «Algol Bulletin» №31 за март 1970 года, труднодоступен, однако Пьер Лекан любезно предоставил доступ к отсканированному файлу на своем веб-сайте.)

Моррис (Morris, 1968, с. 122-124) первым заметил, что при помощи рекурсивных типов можно построить правильно типизированный оператор `fix` для термов (§20.1).

Отношение между изорекурсивной и эквирикурсивной системами исследовалось Абади и Фьоре (Abadi and Fiore, 1996). Два подхода к формализации рекурсивных типов появились уже в ранних работах в этой области, но хорошо запоминающиеся термины «изорекурсивный» и «эквирикурсивный» появились лишь недавно у Крэри, Харпера и Пури (Crary, Harper, and Puri, 1999).

Ссылки на дополнительную литературу относительно рекурсивных типов с подтипами можно найти в разделе 21.12.

Глава 21

Метатеория рекурсивных типов

В главе 20 были введены два альтернативных представления рекурсивных типов: эквирекурсивные типы, которые по определению эквивалентны своим разверткам, и изорекурсивные типы, где эквивалентность явно отмечается при помощи термов `fold` и `unfold`. В этой главе мы разрабатываем теоретические основания процедур проверки типов для эквирекурсивных типов. (Реализация изорекурсивных типов достаточно прямолинейна). Мы будем работать с системой, поддерживающей одновременно и рекурсивные типы, и подтипы, поскольку на практике они часто используются вместе. Система с эквирекурсивными типами, но без подтипов, была бы лишь ненамного проще, поскольку все равно необходимо проверять эквивалентность рекурсивных типов.

В главе 20 мы видели, что подтиповирование эквирекурсивных типов можно интуитивно понимать в терминах бесконечных деревьев вывода подтиповирования, оперирующих бесконечными типами. Нашей задачей будет уточнить эти интуитивные понятия при помощи математического метода *коиндукции* (*cointermination*), и обозначить четкую взаимосвязь между бесконечными деревьями и бесконечными выводами, с одной стороны, и конечными представлениями, с которыми работает реальный алгоритм подтиповирования, с другой.

Мы начнем в §21.1 с рассмотрения базовой теории индуктивных и коиндуктивных определений, а также связанных с ними принципов доказательства. В §21.2 и §21.3 эта общая теория применяется к случаю подтипов, и даются определения как уже знакомого нам отношения подтиповирования на конечных типах, так и его коиндуктивного расширения на случай бесконечных типов. В §21.4 мы несколько отклоняемся от темы и рассматриваем некоторые вопросы, связанные с транзитивностью (которые, как мы уже видели, являются существенным источником трудностей в системах с подтипами). В §21.5 выводятся простые алгоритмы для проверки принадлежности к индуктивно и коиндуктивно определенным множествам; более эффективные алгоритмы рас-

В этой главе рассматривается простое типизированное лямбда-исчисление с подтипами (рис. 15.1), типами-произведениями (11.5) и эквирекурсивными типами. Соответствующий интерпретатор называется `equirec`.

сматриваются в §21.6. В §21.7 эти алгоритмы применяются к важному частному случаю подтипования — «регулярным» бесконечным типам. В §21.8 вводятся μ -типы как конечный способ записи бесконечных типов, и доказывается, что более сложное (однако реализуемое конечным образом) отношение подтипования для μ -типов соответствует обыкновенному коиндуктивному определению подтипования для бесконечных типов. В §21.9 доказывается, что алгоритм проверки подтипования для μ -типов всегда завершается. В §21.10 этот алгоритм сравнивается с другим, разработанным Амадио и Карделли. В §21.11 кратко обсуждаются изорекурсивные типы.

21.1. Индукция и коиндукция

Допустим, мы зафиксировали некоторое *универсальное множество* (universal set) (или *универсум*) \mathcal{U} в качестве области рассмотрения для наших индуктивных и коиндуктивных определений. \mathcal{U} представляет собой множество «всех вещей в мире». Роль индуктивных и коиндуктивных определений будет заключаться в том, чтобы выбрать некоторое подмножество \mathcal{U} . (В дальнейшем в качестве \mathcal{U} мы будем использовать множество всех пар типов, так что подмножества \mathcal{U} будут отношениями на типах. Однако для текущего обсуждения нам подойдет произвольное множество \mathcal{U} .)

Определение 21.1.1 *Функция $F \in \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ монотонна (monotone), если из $X \subseteq Y$ следует, что $F(X) \subseteq F(Y)$. (Напомним, что $\mathcal{P}(\mathcal{U})$ — множество всех подмножеств \mathcal{U} .)*

В дальнейшем мы предполагаем, что F — некоторая монотонная функция на $\mathcal{P}(\mathcal{U})$. Мы часто будем называть F *порождающей функцией* (generating function).

Определение 21.1.2 *Пусть X — подмножество \mathcal{U} .*

1. *X является F -замкнутым (F -closed), если $F(X) \subseteq X$.*
2. *X является F -консistentным (F -consistent), если $X \subseteq F(X)$.*
3. *X является неподвижной точкой (fixed point) F , если $F(X) = X$.*

В этих определениях удобно рассматривать элементы \mathcal{U} как некоторого рода утверждения или высказывания, а F — как отношение «обоснования», которое, получая некоторое множество высказываний (предпосылки), говорит, какие утверждения (заключения) из них следуют. В таком случае, F -замкнутым является такое множество, которое невозможно увеличить, добавляя элементы, обоснованные с помощью F — это множество уже содержит все заключения, которые можно сделать из его членов. F -консistentное множество, соответственно, «самообосновано»: каждое утверждение в нем обосновано другими утверждениями, которые также являются его членами. Неподвижная точка F — множество, одновременно замкнутое и консistentное: оно включает все обоснования, требуемые его членами, все заключения, которые следуют из его членов, и ничего более.

Пример 21.1.3 Рассмотрим следующую порождающую функцию на трехэлементном универсальном множестве $\mathcal{U} = \{a, b, c\}$:

$$\begin{array}{llll} E_1(\emptyset) & = & \{c\} & E_1(\{a, b\}) = \{c\} \\ E_1(\{a\}) & = & \{c\} & E_1(\{a, c\}) = \{b, c\} \\ E_1(\{b\}) & = & \{c\} & E_1(\{b, c\}) = \{a, b, c\} \\ E_1(\{c\}) & = & \{b, c\} & E_1(\{a, b, c\}) = \{a, b, c\} \end{array}$$

Имеется только одно E_1 -замкнутое множество $\{a, b, c\}$ и четыре E_1 -консистентных множества — $\emptyset, \{c\}, \{b, c\}, \{a, b, c\}$

E_1 можно компактно представить набором правил вывода (*inference rules*):

$$\frac{\overline{c} \quad \overline{b}}{c \quad b} \qquad \frac{b \quad c}{a}$$

Каждое из этих правил утверждает, что если все элементы над чертой входят в исходное множество, то элемент под чертой входит в результирующее множество.

Теорема 21.1.4 [КНАСТЕР-ТАРСКИЙ (TARSKI, 1955)]:

1. Пересечение всех F -замкнутых множеств является наименьшей неподвижной точкой F .
2. Объединение всех F -консистентных множеств является наибольшей неподвижной точкой F .

Доказательство: Мы рассматриваем только часть (2); часть (1) доказывается симметричным образом. Пусть $C = \{X \mid X \subseteq F(X)\}$ есть совокупность всех F -консистентных множеств, и пусть P есть объединение всех таких множеств. Учитывая, что F монотонна, и что для всякого $X \in C$ мы знаем, что X F -консистентно и что $X \subseteq P$, получаем $X \subseteq F(X) \subseteq F(P)$. Следовательно, $P = \bigcup_{X \in C} X \subseteq F(P)$, т. е., множество P F -консистентно. Более того, исходя из определения, P — наибольшее F -консистентное множество. Снова используя монотонность F , получаем $F(P) \subseteq F(F(P))$. Это означает, по определению C , что $F(P) \in C$. Следовательно, как и для любого члена C , имеем $F(P) \subseteq P$, т. е., P является F -замкнутым. Мы установили, что P — наибольшее F -консистентное множество, и что P — неподвижная точка F . Таким образом, P — наибольшая неподвижная точка.

Определение 21.1.5 Наименьшая неподвижная точка F записывается в виде μF . Наибольшая неподвижная точка F записывается в виде νF .

Пример 21.1.6 Для вышеприведенной порождающей функции E_1 имеем $\mu E_1 = \nu E_1 = \{a, b, c\}$.

Упражнение 21.1.7 [★]: Допустим, порождающая функция E_2 для универсума $\{a, b, c\}$ определяется следующими правилами вывода:

$$\frac{\overline{a} \quad \overline{c} \quad \overline{b}}{a \quad b} \qquad \frac{a \quad b}{c}$$

Явно выпишите множество пар, составляющих отношение E_2 , как мы это проделали выше для E_1 . Перечислите все E_2 -замкнутые и все E_2 -консистентные множества. Чему равны μE_2 и νE_2 ?

Заметим, что множество μF само по себе F -замкнуто (следовательно, оно является наименьшим F -замкнутым множеством), а νF F -консистентно (следовательно, оно является наибольшим F -консистентным множеством). Это наблюдение дает нам пару основополагающих способов рассуждения:

Следствие 21.1.8 [из 21.1.4]:

1. *Принцип индукции: Если X является F -замкнутым, то $\mu F \subseteq X$.*
2. *Принцип коиндукции: Если X является F -консистентным, то $X \subseteq \nu F$.*

Интуитивно эти принципы можно обосновать так: множество X — это предикат, определенный своим характеристическим множеством — подмножеством \mathcal{U} , для которого предикат верен. Показать, что свойство X выполняется для некоторого элемента x — то же самое, что показать, что x принадлежит множеству X . Принцип индукции говорит, что всякое свойство, характеристическое множество которого замкнуто относительно F (т. е. F сохраняет свойство X), верно для всех элементов индуктивно определенного множества μF .

Соответственно, принцип коиндукции позволяет проверить, принадлежит ли x коиндуктивно определенному множеству νF . Чтобы показать, что $x \in \nu F$, достаточно найти множество X такое, что $x \in X$ и X является F -консистентным. Несмотря на то, что коиндукция несколько менее известна, чем индукция, принцип коиндукции играет важную роль во многих областях информатики; например, он является основным приемом доказательства в теориях параллелизма, основанных на *бисимуляции* (*bisimulation*), а также лежит в основе многих алгоритмов *проверки моделей* (*model checking*).

Принципы индукции и коиндукции широко применяются в этой главе. Мы не выписываем каждое индуктивное рассуждение в виде порождающих функций и предикатов; напротив, мы для краткости зачастую используем уже известные нам инструменты, например, структурную индукцию. Коиндуктивные алгоритмы излагаются подробнее.

Упражнение 21.1.9 [РЕКОМЕНДУЕТСЯ, ★ ★ ★]: Покажите, что принципы обыкновенной индукции на натуральных числах (2.4.1), а также лексикографической индукции по парам натуральных чисел (2.4.4) следуют из принципа индукции 21.1.8

21.2. Конечные и бесконечные типы

Мы намерены конкретизировать общие определения наибольшей неподвижной точки и коиндуктивного метода доказательства подробностями, относящимися к подтипованию. Однако сначала требуется формально показать, как можно рассматривать типы в виде деревьев (конечных либо бесконечных).

Для краткости в этой главе мы имеем дело только с тремя конструкторами типов: \rightarrow , \times и Top . Мы представляем типы как (возможно, бесконечные) деревья, чьи вершины помечены одним из трех символов: \rightarrow , \times или Top . Определения подогнаны к нашим конкретным потребностям; изложение общей теории бесконечных помеченных деревьев можно найти у Курселля (Courcelle, 1983).

Мы используем запись $\{1, 2\}^*$ для обозначения множества последовательностей, состоящих из единиц и двоек. Напомним, что пустая последовательность изображается знаком \bullet , а запись i^k означает k копий i . Если π и σ — последовательности, то π, σ обозначает конкатенацию π и σ .

Определение 21.2.1 Древовидный тип (*tree type*)¹ (или просто дерево, *tree*) есть частичная функция $T \in \{1, 2\}^* \rightarrow \{\rightarrow, \times, \text{Top}\}$, удовлетворяющая следующим ограничениям:

- $T(\bullet)$ определена;
- если определена $T(\pi, \sigma)$, то определена и $T(\pi)$;
- если $T(\pi) = \rightarrow$, либо $T(\pi) = \times$, то определены $T(\pi, 1)$ и $T(\pi, 2)$;
- если $T(\pi) = \text{Top}$, то $T(\pi, 1)$ и $T(\pi, 2)$ не определены.

Древовидный тип T конечен (*finite*), если конечен $\text{dom}(T)$. Множество всех древовидных типов обозначается символом \mathcal{T} ; подмножество конечных типов записывается как \mathcal{T}_f .

Для удобства записи Top будет означать дерево T , где $T(\bullet) = \text{Top}$. Если T_1 и T_2 — деревья, то $T_1 \times T_2$ обозначает такое дерево, для которого $(T_1 \times T_2)(\bullet) = \times$, а $(T_1 \times T_2)(i, \pi) = T_i(\pi)$; $T_1 \rightarrow T_2$ обозначает дерево, для которого $(T_1 \rightarrow T_2)(\bullet) = \rightarrow$, а $(T_1 \rightarrow T_2)(i, \pi) = T_i(\pi)$, для $i = 1, 2$. Например, $(\text{Top} \times \text{Top}) \rightarrow \text{Top}$ обозначает конечный древовидный тип T , определяемый функцией, дающей $T(\bullet) = \rightarrow$, $T(1) = \times$ и $T(2) = T(1, 1) = T(1, 2) = \text{Top}$. С помощью многоточий мы неформально описываем бесконечные древовидные типы. Например, $\text{Top} \rightarrow (\text{Top} \rightarrow (\text{Top} \rightarrow \dots))$ соответствует типу T , у которого $T(2^k) = \rightarrow$ для всех $k \geq 0$, а $T(2^k, 1) = \text{Top}$ для всех $k \geq 0$. Эти соглашения показаны на рис. 21.1.

Множество конечных древовидных типов можно более кратко определить грамматикой:

$$\begin{aligned} T &::= \text{Top} \\ &\quad T \times T \\ &\quad T \rightarrow T \end{aligned}$$

Формально, \mathcal{T}_f — наименьшая неподвижная точка порождающей функции, задаваемой при помощи грамматики. Универсумом этой порождающей функции служит множество всех конечных и бесконечных деревьев, чьи вершины помечены символами Top , \times и \rightarrow (т. е., множество, полученное обобщением определения 21.2.1 через отбрасывание двух последних условий). Все

¹Выражение «древовидный тип» выглядит не слишком красиво, однако его будет очень удобно использовать, когда в §21.8 мы будем обсуждать альтернативное представление рекурсивных типов в виде конечных выражений, содержащих символ μ (μ -типы).

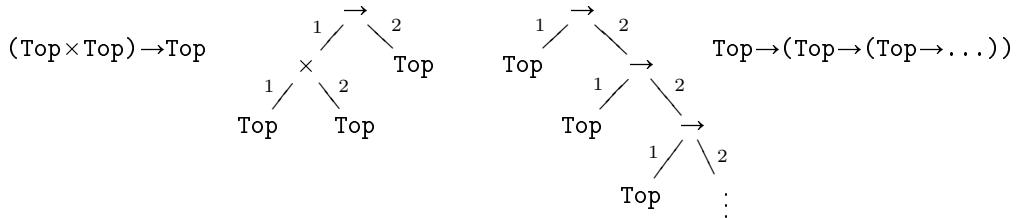


Рис. 21.1. Примеры древовидных типов.

множество \mathcal{T} можно получить из той же самой порождающей функции, взяв наибольшую неподвижную точку вместо наименьшей.

Упражнение 21.2.2 [РЕКОМЕНДУЕТСЯ, ★★]: *Развивая идеи, изложенные в предыдущем абзаце, предложите универсум \mathcal{U} и порождающую функцию $F \in \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ такие, чтобы множество конечных древовидных типов было наименьшей неподвижной точкой F , а множество всех древовидных типов \mathcal{T} — наибольшей неподвижной точкой F .*

21.3. Подтипы

Мы определяем отношения подтиповирования для конечных древовидных типов и для древовидных типов общего вида как, соответственно, минимальную и максимальную неподвижные точки монотонных функций на некоторых универсумах. В случае подтиповирования для конечных древовидных типов в качестве универсума выступает множество $\mathcal{T}_f \times \mathcal{T}_f$ пар конечных древовидных типов; наша порождающая функция будет отображать подмножества этого универсума (то есть, отношения на \mathcal{T}_f) на другие подмножества, и неподвижные точки этой функции также будут отношениями на \mathcal{T}_f . В случае подтиповирования для произвольных (конечных или бесконечных) деревьев универсумом будет $\mathcal{T} \times \mathcal{T}$.

Определение 21.3.1 [Конечное подтиповирование]: *Два конечных древовидных типа S и T находятся в отношении подтиповирования (« S является подтипом T »), если $(S, T) \in \mu S_f$, где монотонная функция $S_f \in \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f) \rightarrow \mathcal{P}(\mathcal{T}_f \times \mathcal{T}_f)$ определяется так:*

$$\begin{aligned} S_f(R) &= \{(T, \text{Top}) \mid T \in \mathcal{T}_f\} \\ &\cup \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1, T_1), (S_2, T_2) \in R\} \\ &\cup \{(S_1 \rightarrow S_2, T_1 \rightarrow T_2) \mid (T_1, S_1), (S_2, T_2) \in R\} \end{aligned}$$

Эта порождающая функция в точности выражает значение обычного опре-

деления отношения подтипования через набор правил вывода:

$$\overline{T \lessdot Top}$$

$$\frac{S_1 \lessdot T_1 \quad S_2 \lessdot T_2}{S_1 \times S_2 \lessdot T_1 \times T_2}$$

$$\frac{T_1 \lessdot S_1 \quad S_2 \lessdot T_2}{S_1 \rightarrow S_2 \lessdot T_1 \rightarrow T_2}$$

Утверждение $S \lessdot T$ над чертой во втором и третьем правилах следует читать как «если пара (S, T) входит в аргумент S_f », а под чертой — как «то (S, T) входит в результат».

Определение 21.3.2 [БЕСКОНЕЧНОЕ ПОДТИПИРОВАНИЕ]: Два (конечных либо бесконечных) древовидных типа S и T находятся в отношении подтипования, если $(S, T) \in \nu S$, где $S \in \mathcal{P}(\mathcal{T} \times \mathcal{T}) \rightarrow \mathcal{P}(\mathcal{T} \times \mathcal{T})$ определяется так:

$$\begin{aligned} S(R) &= \{(T, Top) \mid T \in \mathcal{T}\} \\ &\cup \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1, T_1), (S_2, T_2) \in R\} \\ &\cup \{(S_1 \rightarrow S_2, T_1 \rightarrow T_2) \mid (T_1, S_1), (S_2, T_2) \in R\} \end{aligned}$$

Заметим, что представление этого отношения в виде правил вывода точно такое же, как для вышеприведенного индуктивного отношения; изменения состоят только в том, что мы рассматриваем более широкий универсум типов, и вместо наименьшей выбираем наибольшую неподвижную точку.

Упражнение 21.3.3 [★]: Покажите, что νS не совпадает со всем $\mathcal{T} \times \mathcal{T}$, приведя пример пары (S, T) , не лежащей в νS .

Упражнение 21.3.4 [★]: Существует ли пара типов (S, T) , связанная отношением νS , но не μS ? А пара типов (S, T) , связанная отношением νS_f , но не μS_f ?

Следует сразу проверить базовое свойство отношения подтипования на бесконечных древовидных типах — транзитивность. (В §16.1 мы уже видели, что подтиповование на конечных типах транзитивно.) Если бы отношение подтиповования не было транзитивным, немедленно было бы утрачено крайне важное свойство сохранения типов при вычислении. Чтобы убедиться в этом, предположим, что имеются типы S, T и U , такие, что $S \lessdot T, T \lessdot U$, но неверно, что $S \lessdot U$. Пусть s — значение типа S , а f — функция типа $U \rightarrow Top$. Тогда для терма $(\lambda x:T. f x) s$ можно найти тип, употребив правило включения один раз для каждого применения, однако за один шаг он переходит в неверно типизированный терм $f s$.

Определение 21.3.5 Отношение $R \subseteq \mathcal{U} \times \mathcal{U}$ транзитивно (*transitive*), если оно замкнуто относительно монотонной функции $TR(R) = \{(x, y) \mid \exists z \in \mathcal{U}. (x, z), (z, y) \in R\} — т. е., если $TR(R) \subseteq R$.$

Лемма 21.3.6 Пусть имеется монотонная функция $F \in \mathcal{P}(\mathcal{U} \times \mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U} \times \mathcal{U})$. Если $TR(F(R)) \subseteq F(TR(R))$ для любого $R \subseteq \mathcal{U} \times \mathcal{U}$, то отношение νF транзитивно.

Доказательство: Поскольку νF является неподвижной точкой, то $\nu F = F(\nu F)$, а отсюда $TR(\nu F) = TR(F(\nu F))$. Следовательно, согласно предположению леммы, $TR(\nu F) \subseteq F(TR(\nu F))$. Другими словами, отношение $TR(\nu F)$ F -кonsистентно, а следовательно, по принципу коиндукции, $TR(\nu F) \subseteq \nu F$. Последнее утверждение по определению 21.3.5 эквивалентно транзитивности νF .

Доказательство этой леммы напоминает традиционный метод, которым устанавливают избыточность правила транзитивности в системах логического вывода. Этот метод часто называют «*доказательством посредством устранения сечений*» (cut elimination proofs) (см. §16.1). Условие $TR(F(R)) \subseteq F(TR(R))$ соответствует ключевому шагу в таких доказательствах: если дано, что некоторое утверждение выводимо путем использования некоторых утверждений из R , применения правил из F , а затем правила транзитивности TR , то мы показываем, что можно также изменить порядок применения правил — сначала использовать правило транзитивности, а затем правила из F . С помощью этой леммы мы устанавливаем транзитивность отношения подтипования.

Теорема 21.3.7 Отношение νS транзитивно.

Доказательство: Согласно лемме 21.3.6, достаточно показать, что $TR(S(R)) \subseteq S(TR(R))$ для любого $R \subseteq \mathcal{T} \times \mathcal{T}$. Пусть $(S, T) \in TR(S(R))$. Согласно определению TR , существует некоторый тип $U \in \mathcal{T}$ такой, что $(S, U), (U, T) \in S(R)$. Нам нужно показать, что $(S, T) \in S(TR(R))$. Рассмотрим возможные формы U .

Вариант: $U = \text{Top}$

Поскольку $(U, T) \in S(R)$, из определения S следует, что T должен равняться Top . Однако $(A, \text{Top}) \in S(Q)$ для любых A и Q ; в частности, $(S, T) = (S, \text{Top}) \in S(TR(R))$.

Вариант: $U = U_1 \times U_2$

Если $T = \text{Top}$, то $(S, T) \in S(TR(R))$, как в предыдущем варианте. В противном случае, из $(U, T) \in S(R)$ следует, что $T = T_1 \times T_2$, причем $(U_1, T_1), (U_2, T_2) \in R$. Аналогично, из $(S, U) \in S(R)$ следует, что $S = S_1 \times S_2$, причем $(S_1, U_1), (S_2, U_2) \in R$. Согласно определению TR , имеем $(S_1, T_1), (S_2, T_2) \in TR(R)$, откуда по определению S следует, что $(S_1 \times S_2, T_1 \times T_2) \in S(TR(R))$.

Вариант: $U = U_1 \rightarrow U_2$

Аналогично.

Упражнение 21.3.8 [РЕКОМЕНДУЕТСЯ, ★★]: Покажите, что отношение подтипования на бесконечных деревьях также рефлексивно.

В следующем разделе мы продолжаем обсуждать транзитивность, сравнивая подход к ней в стандартных описаниях конечных типов и в нашем теперешнем изложении теории подтипов для бесконечных древовидных типов. При первом чтении этот раздел можно пропустить или бегло просмотреть.

21.4. Отступление о транзитивности

В главе 16 мы узнали, что стандартные формулировки индуктивно определяемых отношений подтипования, как правило, излагаются в двух формах: *декларативное* представление, оптимизированное с точки зрения читаемости, и *алгоритмическое*, которое более или менее напрямую соответствует реализации. В простых системах эти два представления достаточно похожи друг на друга; в более сложных они могут существенно различаться, и доказательство их эквивалентности может быть непростой задачей. (Пример этого мы увидим в главе 28; существует также множество других примеров.)

Одно из наиболее существенных различий между декларативными и алгоритмическими представлениями состоит в том, что в декларативных представлениях содержится явное правило транзитивности — если $S <: U$ и $U <: T$, то $S <: T$, — а в алгоритмических его нет. Это правило бесполезно для алгоритма, поскольку для его применения при выводе пришлось бы угадывать U .

В декларативных системах правило транзитивности выполняет две полезные функции. Во-первых, оно с очевидностью показывает читателю, что отношение подтипования действительно транзитивно. Во-вторых, правило транзитивности часто позволяет проще и примитивнее сформулировать другие правила; в алгоритмических представлениях приходится сочетать эти простые правила и получать тяжеловесные мегаправила, чтобы учесть все возможные комбинации простых правил. Например, если в системе есть транзитивность, то правила «подтипования в глубину» для полей записей, «подтипования в ширину» путем добавления новых полей и «перестановки» полей можно вводить по отдельности, чтобы их было проще понять; мы сделали так в §15.2. Без транзитивности приходится объединять все эти три правила в одно, учитывающее одновременно глубину, ширину и перестановки, как мы это сделали в §16.1.

До некоторой степени удивительно, что сама возможность построить декларативное представление с правилом транзитивности оказывается следствием «трюка», который можно проделать с индуктивными, но не с коиндуктивными определениями. Чтобы понять причину этого, заметим, что свойство транзитивности есть *свойство замыкания* — оно требует, чтобы отношение подтипования было замкнуто относительно правила транзитивности. Так как отношение подтипования для конечных типов само по себе определяется как замыкание относительно некоторого набора правил, то замыкания транзитивности можно добиться, просто добавив его к набору правил. Таково общее свойство индуктивных определений и свойств замыкания: объединение двух наборов правил, будучи применено индуктивно, порождает наименьшее отношение, замкнутое относительно обоих наборов правил, взятых по отдельности. Это свойство можно сформулировать более абстрактно, в терминах порождающих функций:

Утверждение 21.4.1 Допустим, имеются монотонные функции F и G , и пусть $H(X) = F(X) \cup G(X)$. Тогда μH — наименьшее множество, которое одновременно F -замкнуто и G -замкнуто.

Доказательство: Во-первых, покажем, что μH замкнуто относительно F и G . По определению, $\mu H = H(\mu H) = F(\mu H) \cup G(\mu H)$, так что $F(\mu H) \subseteq \mu H$ и $G(\mu H) \subseteq \mu H$. Во-вторых, покажем, что μH — наименьшее множество, замкнутое относительно одновременно F и G . Допустим, имеется некоторое множество X такое, что $F(X) \subseteq X$ и $G(X) \subseteq X$. Тогда $H(X) = F(X) \cup G(X) \subseteq X$, то есть, X является H -замкнутым. Поскольку μH — наименьшее H -замкнутое множество (по теореме Кнастера-Тарского), имеем $\mu H \subseteq X$.

К сожалению, этот прием для получения транзитивного замыкания не работает в случае коиндуктивных определений. Как показывает следующее упражнение, если добавить транзитивность к правилам, порождающим коиндуктивно определенное отношение, то полученное отношение всегда будет вырождено.

Упражнение 21.4.2 [★]: Допустим, имеется порождающая функция F на универсуме \mathcal{U} . Покажите, что наибольшая неподвижная точка νF^{TR} порождающей функции

$$F^{TR}(R) = F(R) \cup TR(R)$$

представляет собой всюду определенное (*total*) отношение $\mathcal{U} \times \mathcal{U}$.

Поэтому в случае коиндуктивных отношений мы отказываемся от декларативных представлений и работаем только с алгоритмическими.

21.5. Проверка принадлежности

Теперь сконцентрируем внимание на главном вопросе этой главы: как определить при данной порождающей функции F на некотором универсуме \mathcal{U} и элементе $x \in \mathcal{U}$, принадлежит ли x наибольшей неподвижной точке F . Проверка принадлежности в наименьших неподвижных точках рассматривается более кратко (в упр. 21.5.13).

В общем случае данный элемент $x \in \mathcal{U}$ может порождаться функцией F многими различными способами. То есть, может существовать более одного множества $X \subseteq \mathcal{U}$ такого, что $x \in F(X)$. Назовем всякое такое множество *порождающим множеством* (generating set) для x . Поскольку F монотонна, любое надмножество порождающего множества для x также будет порождающим множеством для x , так что имеет смысл говорить только о минимальных порождающих множествах. Делая еще один шаг, мы ограничиваем наше внимание классом «обратимых» порождающих функций, в которых для каждого x имеется не более одного порождающего множества.

Определение 21.5.1 Порождающая функция F называется обратимой (*invertible*), если для каждого $x \in \mathcal{U}$ набор множеств

$$G_x = \{X \subseteq \mathcal{U} \mid x \in F(X)\}$$

либо пуст, либо содержит ровно один член, являющийся подмножеством всех остальных. Если F обратима, то для нее определена частичная функция поддержки $\text{support}_F \in \mathcal{U} \rightarrow \mathcal{P}(\mathcal{U})$:

$$\text{support}_F(x) = \begin{cases} X & \text{если } X \in G_x \text{ и } \forall X' \in G_X. X \subseteq X' \\ \uparrow & \text{если } G_X = \emptyset \end{cases}$$

Функция support определяется также для множеств:

$$\text{support}_F(X) = \begin{cases} \bigcup_{x \in X} \text{support}_F(x) & \text{если } \forall x \in X. \text{support}_F(x) \downarrow \\ \uparrow & \text{в противном случае} \end{cases}$$

Если F ясна из контекста, мы часто будем опускать индекс в названии функции support_F (а также других функций, основанных на F , которые мы определим ниже).

Упражнение 21.5.2 [★★]: Убедитесь, что S_f и S , порождающие функции для отношений подтипования из определений 21.3.1 и 21.3.2, обратимы, и напишите их функции поддержки.

Нашей целью является разработка алгоритмов для проверки принадлежности в наименьшей и наибольшей неподвижной точке порождающей функции F . Элементарные шаги этих алгоритмов будут включать «обратное применение F »: чтобы проверить членство элемента x , требуется узнать, как x мог быть порожден функцией F . Преимущество обратимой F состоит в том, что всегда существует не более одного способа породить данный x . В случае необратимой F элементы могут порождаться различными способами, и это может приводить к комбинаторному взрыву количества путей, которые алгоритм должен проверить. С этого момента мы ограничиваем рассмотрение случаем обратимых порождающих функций.

Определение 21.5.3 Элемент x F -поддерживается (F -supported), если $\text{support}_F(x) \downarrow$; в противном случае x называется F -висячим (F -unsupported). F -поддержivableий элемент называется F -опорным (F -ground), если $\text{support}_F(x) = \emptyset$.

Заметим, что висячий элемент x не входит в $F(X)$ ни для какого X , а опорный x , наоборот, является элементом $F(X)$ для любого X .

Обратимую функцию можно представить в виде *графа поддержки*. Например, на рис. 21.2 функция E на универсуме $\{a, b, c, d, e, f, g, h, i\}$ определяется путем демонстрации того, какие элементы требуются, чтобы поддержать каждый данный элемент универсума: для каждого x множество $\text{support}_E(x)$ содержит все y , к которым от x ведут стрелки. Висячий элемент обозначается перечеркнутым кружком. В этом примере i — единственный висячий элемент, а g — единственный опорный элемент. (Обратите внимание, что h , по нашему определению, поддерживается, несмотря на то, что его множеству поддержки принадлежит висячий элемент.)

Упражнение 21.5.4 [★]: Приведите правила вывода, соответствующие этой функции, как мы это сделали в примере 21.1.3. Убедитесь, что $E(\{b, c\}) =$

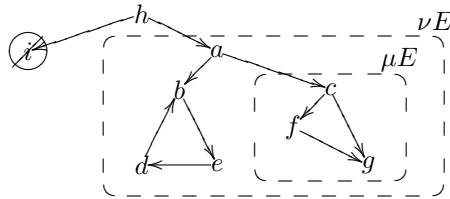


Рис. 21.2. Пример функции *support*.

$\{g, a, d\}$, что $E(\{a, i\}) = \{g, h\}$, и что множества элементов, отмеченные на схеме как μE и νE , действительно являются наименьшей и наибольшей неподвижными точками E .

Взгляд на рис. 21.2 наводит на мысль о том, что элемент x принадлежит к наибольшей неподвижной точке тогда и только тогда, когда из него в графе поддержки недостижим ни один висячий элемент. Отсюда следует алгоритмическая стратегия проверки принадлежности x к множеству νF : нужно перечислить элементы, достижимые из x через функцию *support*; объявить отрицательный результат, если достигнут висячий элемент; иначе объявить положительный результат. Заметим, однако, что между элементами могут иметься циклы достижимости, и процедура перечисления должна позаботиться о том, чтобы не попасть в бесконечный цикл. В оставшейся части раздела мы исследуем эту стратегию.

Определение 21.5.5 Допустим, у нас имеется обратимая порождающая функция F . Определим булевскую функцию gfp_F (или просто gfp ; это сокращение от greatest fixed point — «наибольшая неподвижная точка») следующим образом:²

$$\begin{aligned} gfp(X) &= \text{если } support(X) \uparrow, \text{ то ложь} \\ &\quad \text{иначе если } support(X) \subseteq X, \text{ то истина} \\ &\quad \text{иначе } gfp(support(X) \cup X) \end{aligned}$$

Рассуждая неформально, gfp начинает с X и расширяет его при помощи $support$ до тех пор, пока множество не станет консистентным, либо пока не обнаружится висячий элемент. Определение gfp можно расширить на отдельные элементы, приняв $gfp(x) = gfp(\{x\})$.

Упражнение 21.5.6 [★]: На основании рис. 21.2 можно сделать еще одно наблюдение: элемент x множества νF не входит в μF , если через него проходит цикл в графе поддержки (или если существует путь от x к элементу

²Здесь мы пользуемся стандартным способом определения рекурсивных функций, т. е., считаем, что gfp — наименьшая частичная функция, удовлетворяющая данному уравнению. Такие определения сами по себе можно рассматривать как наименьшие неподвижные точки соответствующих порождающих функций. Подробности можно найти в любом учебнике по денотационной семантике, напр., у Гантера (Gunter, 1992), Уинскела (Winskel, 1993) или Митчелла (Mitchell, 1996).

цикла). Верно ли обратное — то есть, если x является членом νF , но не μF , всегда ли от x есть путь к циклу?

В оставшейся части этого раздела мы доказываем правильность определения функции gfp , а также то, что ее вычисление всегда завершается. (При первом чтении этот материал можно пропустить и сразу перейти к следующему разделу.) Для начала установим некоторые свойства функции $support$.

Лемма 21.5.7 $X \subseteq F(Y)$ тогда и только тогда, когда $support_F(X) \downarrow$ и $support_F(X) \subseteq Y$.

Доказательство: Достаточно показать, что $x \in F(Y)$ тогда и только тогда, когда $support(x) \downarrow$ и $support(x) \subseteq Y$. Предположим вначале, что $x \in F(Y)$. Тогда $Y \in G_x = \{X \subseteq \mathcal{U} \mid x \in F(X)\}$ — т. е., $G_x \neq \emptyset$. Следовательно, поскольку F обратима, $support(x)$, наименьший элемент G_x , существует, и $support(x) \subseteq Y$. В обратном направлении, если $support(x) \subseteq Y$, то $F(support(x)) \subseteq F(Y)$ из-за монотонности F . Однако $x \in F(support(x))$ согласно определению функции $support$, так что $x \in F(Y)$.

Лемма 21.5.8 Допустим, P — неподвижная точка функции F . Тогда $X \subseteq P$ тогда и только тогда, когда $support_F(X) \downarrow$ и $support_F(X) \subseteq P$.

Доказательство: Достаточно вспомнить, что $P = F(P)$, и применить лемму 21.5.7.

Теперь мы можем доказать правильность определения gfp как частичной функции. (Полная правильность нас пока не интересует, поскольку при некоторых порождающих функциях gfp будет расходиться. Далее в этом разделе мы докажем, что для ограниченного класса порождающих функций вычисление gfp всегда завершается.)

Теорема 21.5.9

1. Если $gfp_F(X) =$ истина, то $X \subseteq \nu F$.

2. Если $gfp_F(X) =$ ложь, то $X \not\subseteq \nu F$.

Доказательство: Доказательство каждого из утверждений проводится индукцией по рекурсивной структуре вызовов алгоритма.

1. Из определения gfp легко видеть, что эта функция может вернуть значение истина в двух случаях. Если $gfp(X) =$ истина потому, что $support(X) \subseteq X$, то, по лемме 21.5.7, мы имеем $X \subseteq F(X)$, т. е., множество X является F -кonsистентным; таким образом, $X \subseteq \nu F$ по принципу коиндукции. Если же, с другой стороны, $gfp(X) =$ истина потому, что $gfp(support(X) \cup X) =$ истина, то, согласно предположению индукции, $support(X) \cup X \subseteq \nu F$ и, таким образом, $X \subseteq \nu F$.

2. Опять же, есть два способа получить $gfp(X) =$ ложь. Сначала предположим, что $gfp(X) =$ ложь потому, что $support(X) \uparrow$. Тогда $X \not\subseteq \nu F$ по лемме 21.5.8. Предположим, с другой стороны, что $gfp(X) =$ ложь

потому, что $gfp(support(X) \cup X) = \text{ложь}$. Согласно предположению индукции, $support(X) \cup X \not\subseteq \nu F$. Это утверждение равносильно $X \not\subseteq \nu F$ либо $support(X) \not\subseteq \nu F$. В любом случае, $X \not\subseteq \nu F$ (во втором варианте — по лемме 21.5.8).

Теперь мы должны найти достаточное условие завершения gfp , и таким образом получить класс порождающих функций, для которых алгоритм определения принадлежности гарантированно завершается. Для описания этого класса нам потребуется некоторая дополнительная терминология.

Определение 21.5.10 Если имеется обратимая порождающая функция F и элемент $x \in \mathcal{U}$, множество $\text{pred}_F(x)$ (или просто $\text{pred}(x)$) непосредственных предшественников x определяется как

$$\text{pred}(x) = \begin{cases} \emptyset & \text{если } support(x) \uparrow \\ support(x) & \text{если } support(x) \downarrow \end{cases}$$

а расширение этого понятия на множество $X \subseteq \mathcal{U}$ — как

$$\text{pred}(X) = \bigcup_{x \in X} \text{pred}(x).$$

Множество $\text{reachable}_F(X)$ (или просто $\text{reachable}(X)$) элементов, достижимых из множества X через $support$, определяется как

$$\text{reachable}(X) = \bigcup_{n \geq 0} \text{pred}^n(X).$$

а расширение этого понятия на отдельные элементы $x \in \mathcal{U}$ — как

$$\text{reachable}(x) = \text{reachable}(\{x\}).$$

Элемент $y \in \mathcal{U}$ достижим (reachable) из элемента x , если $y \in \text{reachable}(x)$.

Определение 21.5.11 Обратимая порождающая функция F называется функцией с конечным множеством состояний (*finite state*), если множество $\text{reachable}(x)$ конечно для любого $x \in \mathcal{U}$.

Для порождающей функции с конечным множеством состояний пространство поиска, просматриваемое алгоритмом gfp , конечно, и gfp всегда завершается:

Теорема 21.5.12 Если $\text{reachable}_F(X)$ конечно, то $gfp_F(X)$ определена. Следовательно, если F — функция с конечным множеством состояний, то $gfp_F(X)$ завершается для всякого конечного $X \subseteq \mathcal{U}$.

Доказательство: Для каждого рекурсивного вызова $gfp(Y)$ в графе вызовов, порожденном исходным вызовом $gfp(X)$, имеем $Y \subseteq \text{reachable}(X)$. Более того, множество Y при каждом вызове строго увеличивается. Поскольку множество $\text{reachable}(X)$ конечно, $t(Y) = |\text{reachable}(X)| - |Y|$ можно использовать как меру завершения для gfp .

Упражнение 21.5.13 [★★★]: Допустим, имеется обратимая порождающая функция F . Определим функцию lfp_F (или просто lfp) как

$$\begin{aligned}\text{lfp}(X) &= \text{если } \text{support}(X) \uparrow, \text{ то ложь} \\ &\quad \text{иначе если } X = \emptyset, \text{ то истина} \\ &\quad \text{иначе } \text{lfp}(\text{support}(X)).\end{aligned}$$

Неформально, lfp начинает с множества X и уменьшает его до тех пор, пока оно не станет пустым. Докажите частичную правильность этого алгоритма в том смысле, что

1. Если $\text{lfp}_F(X) = \text{истина}$, то $X \subseteq \mu F$.
2. Если $\text{lfp}_F(X) = \text{ложь}$, то $X \not\subseteq \mu F$.

Можно ли указать класс порождающих функций, для которых lfp_F гарантированно завершается для всех конечных входов?

21.6. Более эффективные алгоритмы

Хоть алгоритм gfp дает правильный результат, он не очень эффективен, поскольку требуется перевычислять значение support для всего множества X при каждом рекурсивном вызове. Например, в следующей последовательности вызовов gfp для функции E с рис. 21.2,

$$\begin{aligned}&\text{gfp}(\{a\}) \\ &= \text{gfp}(\{a, b, c\}) \\ &= \text{gfp}(\{a, b, c, e, f, g\}) \\ &= \text{gfp}(\{a, b, c, e, f, g, d\}) \\ &= \text{истина}.\end{aligned}$$

Обратите внимание, что $\text{support}(a)$ вычисляется четыре раза. Можно усовершенствовать алгоритм и избавиться от излишних перевычислений, если поддерживать множество *гипотез* (assumptions) A , для элементов которого множества поддержки уже рассмотрены, и множество *целей* (goals) X , для элементов которого множества поддержки еще не рассмотрены.

Определение 21.6.1 Пусть имеется обратимая порождающая функция F . Определим функцию gfp_F^a (или просто gfp^a) следующим образом (верхний индекс a означает assumptions — «гипотезы»):

$$\begin{aligned}\text{gfp}^a(A, X) &= \text{если } \text{support}(X) \uparrow, \text{ то ложь} \\ &\quad \text{иначе если } X = \emptyset, \text{ то истина} \\ &\quad \text{иначе } \text{gfp}^a(A \cup X, \text{support}(X) \setminus (A \cup X))\end{aligned}$$

Чтобы проверить утверждение $x \in \nu F$, достаточно вычислить $\text{gfp}^a(\emptyset, \{x\})$.

Этот алгоритм (как и два других алгоритма из этого раздела) вычисляет множество поддержки для каждого элемента только один раз. Вычисление предыдущего примера выглядит так:

$$\begin{aligned} & gfp^a(\emptyset, \{a\}) \\ = & gfp^a(\{a\}, \{b, c\}) \\ = & gfp^a(\{a, b, c\}, \{e, f, g\}) \\ = & gfp^a(\{a, b, c, e, f, g, d\}, \emptyset) \\ = & \text{истина.} \end{aligned}$$

Естественно, утверждение о корректности этого алгоритма выглядит немного сложнее, чем то, что мы видели в предыдущем разделе.

Теорема 21.6.2

1. Если $support_F(A) \subseteq A \cup X$ и $gfp_F^a(A, X) = \text{истина}$, то $A \cup X \subseteq \nu F$.
2. Если $gfp_F^a(A, X) = \text{ложь}$, то $X \not\subseteq \nu F$.

Доказательство: Аналогично 21.5.9.

В оставшейся части раздела рассматриваются еще два варианта алгоритма gfp , которые более точно соответствуют известным алгоритмам проверки подтипования для рекурсивных типов. При первом чтении остаток этого раздела можно пропустить и перейти к началу следующего раздела.

Определение 21.6.3 Небольшая вариация на тему gfp^a состоит в том, чтобы на каждом шаге брать только один элемент X и раскрывать его функцию $support$. Этот новый алгоритм называется gfp_F^s (или просто gfp^s ; буква s означает *single* — «одиночный»).

$$\begin{aligned} gfp^s(A, X) = & \text{ если } X = \emptyset, \text{ то } \text{истина} \\ & \text{иначе пустъ } x \text{ — некоторый элемент } X, \text{ для которого} \\ & \quad \text{если } x \in A, \text{ то } gfp^s(A, X \setminus \{x\}) \\ & \quad \text{иначе если } support(x) \uparrow, \text{ то } \text{ложь} \\ & \quad \text{иначе } gfp^s(A \cup \{x\}, (X \cup support(x)) \setminus (A \cup \{x\})). \end{aligned}$$

Утверждение о корректности (т. е., инвариант рекурсивного «цикла») для этого алгоритма в точности такое же, как в теореме 21.6.2.

В отличие от вышеупомянутого алгоритма, многие существующие алгоритмы для определения рекурсивного подтипования принимают в качестве аргумента всего один элемент, а не их множество. Еще одна небольшая модификация нашего алгоритма приближает его к таким процедурам. Измененный алгоритм больше не обладает свойством хвостовой рекурсии,³ поскольку стек

³Хвостовым вызовом (tail call) называется рекурсивный вызов, являющийся последним действием вызывающей функции — т. е. такой, что результат, возвращаемый из рекурсивного вызова, будет результатом и вызывающей функции. Хвостовые вызовы представляют интерес потому, что большинство компиляторов для функциональных языков реализуют их как простые переходы, и при этом заново используется место на стеке, занятое вызывающей функцией, а не выделяется новый кадр стека. Вследствие этого цикл, реализованный в виде хвосто-рекурсивной функции, компилируется в такой же машинный код, как эквивалентный ему цикл *while*.

вызовов используется для хранения еще не проверенных подцелей. Еще одно изменение состоит в том, что алгоритм принимает множество гипотез A и возвращает новое множество гипотез в качестве результата. Это позволяет запоминать гипотезы о подтиповании, порожденные в уже завершившихся рекурсивных вызовах, и снова их использовать в последующих вызовах. В сущности, множество гипотез «прошивается» (*is threaded*) сквозь граф рекурсивных вызовов; отсюда название нового алгоритма, gfp^t .

Определение 21.6.4 Если дана обратимая порождающая функция F , то функция gfp_F^t (или просто gfp^t ; буква t означает *threaded* — «прошитый») определяется следующим образом:

$$\begin{aligned} gfp^t(A, x) &= \text{если } x \in A, \text{ то } A \\ &\quad \text{иначе если } support(x) \uparrow, \text{ то неудача} \\ &\quad \text{иначе} \\ &\quad \quad \text{пусть } \{x_1, \dots, x_n\} = support(x), \text{ и} \\ &\quad \quad \text{пусть } A_0 = A \cup \{x\}, \text{ и} \\ &\quad \quad \text{пусть } A_1 = gfp^t(A_0, x_1), \text{ и} \\ &\quad \quad \dots \\ &\quad \quad \text{пусть } A_n = gfp^t(A_{n-1}, x_n), \text{ и} \\ &\quad \quad \text{тогда результатом будет } A_n. \end{aligned}$$

Чтобы проверить утверждение $x \in \nu F$, требуется вычислить $gfp^t(\emptyset, x)$. Если это вычисление завершается успешно, то $x \in \nu F$. Если оно неудачно, то $x \notin \nu F$. Мы используем для неудач следующее соглашение: если вычисление выражения B неудачно, то «пусть $A = B$, и тогда результатом будет C » также неудачно. Таким образом мы избегаем явных блоков «обработки исключений» в каждом рекурсивном вызове gfp^t .

Утверждение о корректности для этого алгоритма требуется еще раз изменить по сравнению с тем, что мы видели выше, на этот раз приняв во внимание его не-хвосторекурсивную формулировку и постулировав дополнительный «стек» X элементов, поддержку которых предстоит проверить.

Лемма 21.6.5

1. Если $gfp_F^t(A, x) = A'$, то $A \cup \{x\} \subseteq A'$.
2. Для каждого X , если $support_F(A) \subseteq A \cup X \cup \{x\}$ и $gfp_F^t(A, x) = A'$, то $support_F(A') \subseteq A' \cup X$.

Доказательство: Часть (1) представляет собой простую индукцию по рекурсивной структуре вызовов алгоритма.

Часть (2) также доказывается индукцией по рекурсивной структуре вызовов алгоритма. Если $x \in A$, то $A' = A$, и требуемый вывод следует из предположения немедленно. С другой стороны, предположим, что $A' \neq A$, и рассмотрим частный случай, когда $support(x)$ содержит два элемента x_1 и x_2 . Общий случай, здесь не показанный, доказывается аналогично с помощью внутренней индукции по размеру $support(x)$. Алгоритм вычисляет A_0 ,

A_1 и A_2 , и возвращает A_2 . Нужно показать для произвольного X_0 , что если $\text{support}(A) \subseteq A \cup \{x\} \cup X_0$, то $\text{support}(A_2) \subseteq A_2 \cup X_0$. Пусть $X_1 = X_0 \cup \{x_2\}$. Поскольку

$$\begin{aligned}\text{support}(A_0) &= \text{support}(A) \cup \text{support}(x) \\ &= \text{support}(A) \cup \{x_1, x_2\} \\ &\subseteq A \cup \{x\} \cup X_0 \cup \{x_1, x_2\} \\ &= A_0 \cup X_0 \cup \{x_1, x_2\} \\ &= A_0 \cup X_1 \cup \{x_1\},\end{aligned}$$

то мы можем применить предположение индукции к первому рекурсивному вызову, конкретизировав универсально квантифицированную переменную X значением X_1 . При этом получаем $\text{support}(A_1) \subseteq A_1 \cup X_1 = A_1 \cup \{x_2\} \cup X_0$. Теперь можно применить предположение индукции ко второму рекурсивному вызову, конкретизировав переменную X значением X_0 и получить требуемый результат: $\text{support}(A_2) \subseteq A_2 \cup X_0$.

Теорема 21.6.6

1. Если $\text{gfp}_F^t(\emptyset, x) = A'$, то $x \in \nu F$.
2. Если $\text{gfp}_F^t(\emptyset, x) = \text{неудача}$, то $x \notin \nu F$.

Доказательство: В части (1) заметим, что по лемме 21.6.5(1), $x \in A'$. Конкретизируя часть (2) леммы через $X = \emptyset$, получаем $\text{support}(A') = A'$ — т. е., множество A' является F -консистентным согласно лемме 21.5.7, и поэтому $A' \subseteq \nu F$ по коиндукции. В части (2) мы рассуждаем (применив несложную индукцию по глубине вызовов при вычислении алгоритма gfp_F^t с использованием леммы 21.5.8), что если для некоторого A мы имеем $\text{gfp}_F^t(A, x) = \text{неудача}$, то $x \notin \nu F$.

Поскольку все алгоритмы этого раздела занимаются просмотром множества достижимых элементов, достаточное условие завершения для всех этих алгоритмов такое же, как и для исходного алгоритма gfp : они завершаются при любом входе, если функция F имеет конечное множество состояний.

21.7. Регулярные деревья

К этому моменту мы разработали общие алгоритмы проверки принадлежности множеству, определенному как наибольшая неподвижная точка порождающей функции F , в предположении, что F обратима и имеет конечное число состояний; отдельно мы показали, как определить подтипование на бесконечных деревьях в виде наибольшей неподвижной точки некоторой порождающей функции S . Очевидным следующим шагом будет конкретизация одного из наших алгоритмов при его применении к S . Разумеется, этот конкретный алгоритм будет завершаться не при всех входных данных, поскольку в общем случае множество состояний, достижимых из некоторой пары бесконечных типов, может быть бесконечным. Однако, как мы увидим в этом разделе, если мы ограничим рассмотрение бесконечными типами некоторой разумной

формы — так называемыми *регулярными типами* (regular types), то получим гарантию того, что множество достижимых состояний будет конечно, и алгоритм проверки подтипования всегда будет завершаться.

Определение 21.7.1 Древовидный тип S является поддеревом (*subtree*) древовидного типа T , если $S = \lambda\sigma.T(\pi, \sigma)$ для некоторого π — то есть, если функцию S из путей в символы можно получить из функции T путем добавления некоторого фиксированного префикса π к путям-аргументам, которые мы передаем функции T ; префикс π соответствует пути от корня дерева T к корню дерева S . Множество всех поддеревьев типа T мы обозначаем как $\text{subtrees}(T)$.

Определение 21.7.2 Древовидный тип $T \in \mathcal{T}$ называется регулярным (*regular*), если множество $\text{subtrees}(T)$ конечно — т. е. если у T конечное число различных поддеревьев. Множество регулярных типов обозначается как \mathcal{T}_r .

Примеры 21.7.3

1. Всякий конечный древовидный тип регулярен; число его различных поддеревьев не больше числа вершин. Число различных поддеревьев также может быть строго меньше числа вершин. Например, у типа $T = \text{Top} \rightarrow (\text{Top} \times \text{Top})$ пять вершин, но всего три различных поддерева (само T , $\text{Top} \times \text{Top}$ и Top).
2. Некоторые бесконечные древовидные типы регулярны. Например, дерево
$$T = \text{Top} \times (\text{Top} \times \dots)$$
имеет всего два различных поддерева — само T и Top .

3. Древовидный тип

$$T = B \times (A \times (B \times (A \times (A \times (B \times (A \times (A \times (B \times \dots)))))))$$

где пары последовательных вхождений B разделяются все большим количеством вхождений A , не является регулярным. Поскольку T не регулярен, множество $\text{reachable}_S(T, T)$, содержащее все пары утверждений о подтиповании, требуемые для доказательства $T <: T$, бесконечно.

Утверждение 21.7.4 Ограничение S_r порождающей функции S на множество регулярных древовидных типов имеет конечное число состояний.

Доказательство: Требуется показать, что для всякой пары регулярных древовидных типов (S, T) , множество $\text{reachable}_{S_r}(S, T)$ конечно. Заметим, что $\text{reachable}_{S_r}(S, T) \subseteq \text{subtrees}(S) \times \text{subtrees}(T)$; последнее множество конечно, поскольку конечны $\text{subtrees}(S)$ и $\text{subtrees}(T)$.

Это означает, что мы можем получить процедуру проверки для отношения подтипования на регулярных древовидных типах, конкретизировав функцией S любой из алгоритмов проверки принадлежности. Разумеется, чтобы это работало в какой-либо практической реализации, нужно представлять регулярные деревья с помощью каких-то конечных структур. Один из вариантов такого представления, μ -нотация, обсуждается в следующем разделе.

21.8. μ -типы

В этом разделе разрабатывается конечная μ -нотация, определяется подтипованием на μ -выражениях, и устанавливается соответствие между ним и подтипованием для древовидных типов.

Определение 21.8.1 Пусть X будет обозначением для любой из счетного множества типовых переменных $\{X_1, X_2, \dots\}$. Множество $\mathcal{T}_m^{\text{raw}}$ сырых μ -типов (*raw μ -types*) есть множество выражений, определяемых следующей грамматикой:

$$\begin{array}{ll} T & ::= \quad X \\ & \quad \text{Top} \\ & \quad T \times T \\ & \quad T \rightarrow T \\ & \quad \mu X. \, T \end{array}$$

Синтаксический оператор μ обозначает связывание, и через него стандартным образом определяются понятия свободной и связанной переменной, замкнутых сырых μ -типов, а также эквивалентности сырых μ -типов с точностью до переименования связанных переменных. Запись $FV(T)$ обозначает множество свободных переменных сырого μ -типа T . Подстановка $[X \mapsto S]T$ с избеганием захвата связанных переменных сырого μ -типа S в сырой μ -тип T также определяется как обычно.

Чтобы достичь должного соответствия с регулярными деревьями, требуется несколько ограничить множество сырых μ -типов: нам хотелось бы «считывать» древовидный тип как бесконечную развертку данного μ -типа, однако существуют сырые μ -типы, которые невозможно разумным образом интерпретировать как представления древовидных типов. В этих типах встречаются подвыражения вида $\mu X. \mu X_1 \dots \mu X_n . X$, где все переменные с X_1 по X_n отличны от X . Рассмотрим, например, $T = \mu X. X$. Развертка T снова дает T , так что с помощью такой операции никакое дерево «считать» невозможно. Это приводит нас к следующему ограничению.

Определение 21.8.2 Сырой μ -тип T называется сократимым (*contractive*), если для любого подвыражения T вида $\mu X. \mu X_1 \dots \mu X_n . S$ тело S не равняется X . Эквивалентное определение: сырой μ -тип сократим, если всякое вхождение μ -связанной переменной в теле отделено от связывающей конструкции, но меньшей мере, одним элементом \rightarrow или \times .

Сырой μ -тип называется просто μ -типов (*μ -type*), если он сократим. Множество μ -типов обозначается \mathcal{T}_m .

Если T — μ -тип, то $\mu\text{-height}(T)$ обозначает количество μ -связываний в его начале.

Обычное понимание μ -типов как конечного представления для бесконечных регулярных деревьев формализуется при помощи следующей функции.

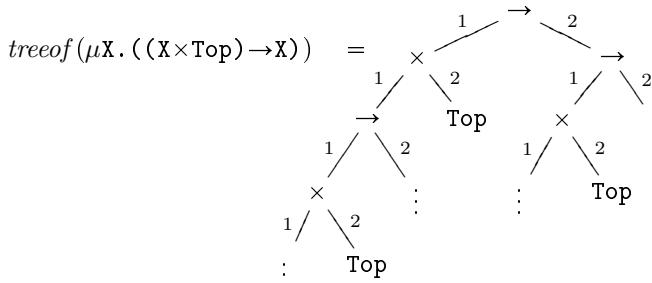


Рис. 21.3. Пример применения функции `treeof`

Определение 21.8.3 Функция `treeof`, отображающая замкнутые μ -типы на древовидные типы, индуктивно определяется так:

$$\begin{aligned}
 \text{treeof}(\text{Top})(\bullet) &= \text{Top} \\
 \text{treeof}(T_1 \rightarrow T_2)(\bullet) &= \rightarrow \\
 \text{treeof}(T_1 \rightarrow T_2)(i, \pi) &= \text{treeof}(T_i)(\pi) \\
 \text{treeof}(T_1 \times T_2)(\bullet) &= \times \\
 \text{treeof}(T_1 \times T_2)(i, \pi) &= \text{treeof}(T_i)(\pi) \\
 \text{treeof}(\mu X. T)(\pi) &= \text{treeof}([X \mapsto \mu X. T] T)(\pi)
 \end{aligned}$$

Чтобы убедиться в том, что это определение корректно (то есть учитывает все варианты, и всегда обеспечивает завершение вычисления функции), заметим следующее:

1. Всякое рекурсивное использование `treeof` в правой части определения уменьшает лексикографический размер пары $(|\pi|, \mu\text{-height}(T))$: варианты для $S \rightarrow T$ и $S \times T$ уменьшают $|\pi|$; вариант для $\mu X. T$ оставляет $|\pi|$ неизменным, но уменьшает $\mu\text{-height}(T)$.
2. Все рекурсивные вызовы сохраняют сократимость и замыкание типовых аргументов. В частности, тип $\mu X. T$ сократим и замкнут тогда и только тогда, когда сократима и замкнута его развертка $[X \mapsto \mu X. T] T$. Это служит обоснованием шага развертки в определении `treeof` ($\mu X. T$).

Определение функции `treeof` распространяется на пары типов: $\text{treeof}(S, T) = (\text{treeof}(S), \text{treeof}(T))$.

Пример применения `treeof` к μ -типу приведен на рис. 21.3.

Отношение подтипов для древовидных типов было определено в §21.3 как наибольшая неподвижная точка порождающей функции S . В этом

разделе мы расширили синтаксис типов μ -типами, поведение которых интуитивно описывается правилами (соответственно, правой и левой) μ -свертки:

$$\frac{S <: [X \mapsto \mu X. T]T}{S <: \mu X. T} \quad \text{и} \quad \frac{[X \mapsto \mu X. S]S <: T}{\mu X. S <: T}$$

Формально мы определяем подтипование для μ -типов через порождающую функцию S_m , в определении которой три варианта совпадают с определением S , а еще два соответствуют правилам μ -свертки.

Определение 21.8.4 Два μ -типа S и T находятся в отношении подтипования, если $(S, T) \in \nu S_m$, где монотонная функция $S_m \in \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m) \rightarrow \mathcal{P}(\mathcal{T}_m \times \mathcal{T}_m)$ определяется уравнением

$$\begin{aligned} S_m(R) = & \{(S, \text{Top}) \mid S \in \mathcal{T}_m\} \\ & \cup \{(S_1 \times S_2, T_1 \times T_2) \mid (S_1, T_1), (S_2, T_2) \in R\} \\ & \cup \{(S_1 \rightarrow S_2, T_1 \rightarrow T_2) \mid (T_1, S_1), (S_2, T_2) \in R\} \\ & \cup \{(S, \mu X. T) \mid (S, [X \mapsto \mu X. T]T) \in R\} \\ & \cup \{(\mu X. S, T) \mid ([X \mapsto \mu X. S]S, T) \in R, T \neq \text{Top} \text{ и } T \neq \mu Y. T_1\} \end{aligned}$$

Заметим, что это определение не в точности соответствует вышеприведенным правилам μ -свертки: мы ввели асимметрию между последним и предпоследним вариантом, чтобы сделать функцию обратимой (иначе варианты перекрывались бы). Однако, как показывает следующее упражнение, S_m порождает то же самое отношение подтипования, что и более естественная порождающая функция⁴ S_d , у которой компоненты определения прямо соответствуют правилам вывода.

Упражнение 21.8.5 [★★★]: Выпишите определение вышеупомянутой функции S_d и покажите, что она не является обратимой. Докажите, что $\nu S_d = \nu S_m$.

Порождающая функция S_m обратима, поскольку определена соответствующая функция поддержки:

$$support_{S_m}(S, T) = \left\{ \begin{array}{ll} \emptyset & \text{если } T = \text{Top} \\ \{(S_1, T_1), (S_2, T_2)\} & \text{если } S = S_1 \times S_2 \text{ и} \\ & T = T_1 \times T_2 \\ \{(T_1, S_1), (S_2, T_2)\} & \text{если } S = S_1 \rightarrow S_2 \text{ и} \\ & T = T_1 \rightarrow T_2 \\ \{(S, [X \mapsto \mu X. T_1]T_1)\} & \text{если } T = \mu X. T_1 \\ \{([X \mapsto \mu X. S_1]S_1, T)\} & \text{если } S = \mu X. S_1 \text{ и} \\ & T \neq \mu X. T_1, T \neq \text{Top} \\ \uparrow & \text{в остальных случаях} \end{array} \right.$$

Отношение подтипования на μ -типах пока что было введено отдельно от определенного ранее отношения подтипования для древовидных типов.

⁴Буква d в S_d напоминает, что эта функция основана на «декларативных» (*declarative*) правилах вывода для μ -свертки, в отличие от «алгоритмических» правил, используемых в S_m .

Поскольку мы рассматриваем μ -типы всего лишь как способ представления регулярных типов в конечном виде, необходимо убедиться, что эти два отношения соответствуют друг другу. Следующая теорема (21.8.7) устанавливает это соответствие. Однако сначала требуется доказать техническую лемму.

Лемма 21.8.6 *Допустим, отношение $R \subseteq \mathcal{T}_m \times \mathcal{T}_m$ является S_m -консистентным. Для любой пары $(S, T) \in R$ имеется некоторая пара $(S', T') \in R$ такая, что $\text{treeof}(S', T') = \text{treeof}(S, T)$, и ни S' , ни T' не начинаются с μ .*

Доказательство: Индукция по сумме количества вхождений μ в начале S и T . Если ни S , ни T не начинаются с μ , то можно взять $(S', T') = (S, T)$. С другой стороны, если $(S, T) = (S, \mu X. T_1)$, то, исходя из S_m -консистентности R , имеем $(S, T) \in S_m(R)$, так что $(S'', T'') = (S, [X \mapsto \mu X. T_1] T_1) \in R$. Так как T сократим, то T'' , результат развертки T , содержит в начале на одну μ меньше, чем T . Согласно предположению индукции, существует некоторая пара типов (S', T') , такая, что ни S' , ни T' не начинаются с μ , и $\text{treeof}(S'', T'') = \text{treeof}(S', T')$. Поскольку, по определению treeof , $\text{treeof}(S, T) = \text{treeof}(S'', T'')$, пара (S', T') — то, что нам нужно. Вариант с $(S, T) = (\mu X. S_1, T)$ доказывается аналогично.

Теорема 21.8.7 *Пусть $(S, T) \in \mathcal{T}_m \times \mathcal{T}_m$. Тогда $(S, T) \in \nu S_m$ тогда и только тогда, когда $\text{treeof}(S, T) \in \nu S$.*

Доказательство: Рассмотрим сначала направление «только если» — покажем, что из $(S, T) \in \nu S_m$ следует $\text{treeof}(S, T) \in \nu S$. Пусть $(A, B) = \text{treeof}(S, T) \in \mathcal{T} \times \mathcal{T}$. Согласно принципу коиндукции, требуемый результат будет получен, если мы продемонстрируем S -консистентное множество $\mathcal{Q} \subseteq \mathcal{T} \times \mathcal{T}$ такое, что $(A, B) \in \mathcal{Q}$. Мы утверждаем, что таким множеством является $\mathcal{Q} = \text{treeof}(\nu S_m)$. Чтобы убедиться в этом, нужно показать, что $(A', B') \in S(\mathcal{Q})$ для всех $(A', B') \in \mathcal{Q}$.

Пусть имеется пара μ -типов $(S', T') \in \nu S_m$, такая, что $\text{treeof}(S', T') = (A', B')$. Исходя из леммы 21.8.6, можно предположить, что ни S' , ни T' не начинаются с μ . Поскольку множество νS_m S_m -консистентно, пара (S', T') должна быть поддержана одним из вариантов в определении S_m , т. е. должна иметь одну из следующих форм:

Вариант: $(S', T') = (S', \text{Top})$

Тогда $B' = \text{Top}$, и $(A', B') \in S(\mathcal{Q})$ по определению S .

Вариант: $(S', T') = (S_1 \times S_2, T_1 \times T_2)$, где $(S_1, T_1), (S_2, T_2) \in \nu S_m$

Согласно определению treeof , имеем $B' = \text{treeof}(T') = B_1 \times B_2$, где каждый $B_i = \text{treeof}(T_i)$. Аналогично, $A' = A_1 \times A_2$, где $A_i = \text{treeof}(S_i)$. Применение функции treeof к этим парам дает $(A_1, B_1), (A_2, B_2) \in \mathcal{Q}$. Но тогда по определению S мы получаем $(A, B) = (A_1 \times A_2, B_1 \times B_2) \in S(\mathcal{Q})$.

Вариант: $(S', T') = (S_1 \rightarrow S_2, T_1 \rightarrow T_2)$, где $(T_1, S_1), (S_2, T_2) \in \nu S_m$

Аналогично.

Теперь проверим направление «если» — покажем, что из $\text{treeof}(S, T) \in \nu S$ следует, что $(S, T) \in \nu S_m$. Согласно принципу коиндукции, достаточно продемонстрировать S_m -консистентное множество $R \in \mathcal{T}_m \times \mathcal{T}_m$ такое,

что $(S, T) \in R$. Мы утверждаем, что таким множеством является $R = \{(S', T') \in \mathcal{T}_m \times \mathcal{T}_m \mid \text{treeof}(S', T') \in \nu S\}$. Ясно, что $(S, T) \in R$. Чтобы завершить доказательство, остается показать, что из $(S', T') \in R$ следует $(S', T') \in S_m(R)$.

Заметим, что поскольку νS является S -консистентным, всякая пара $(A', B') \in \nu S$ должна иметь одну из трех форм: (A', Top) , $(A_1 \times A_2, B_1 \times B_2)$ или $(A_1 \rightarrow A_2, B_1 \rightarrow B_2)$. Отсюда и из определения *treeof* мы видим, что всякая пара $(S', T') \in R$ должна иметь одну из форм (S', Top) , $(S_1 \times S_2, T_1 \times T_2)$, $(S_1 \rightarrow S_2, T_1 \rightarrow T_2)$, $(S', \mu X. T_1)$ либо $(\mu X. S_1, T')$. Рассмотрим все эти варианты поочереди.

Вариант: $(S', T') = (S', \text{Top})$

В этом случае $(S', T') \in S_m(R)$ непосредственно по определению S_m .

Вариант: $(S', T') = (S_1 \times S_2, T_1 \times T_2)$

Пусть $(A', B') = \text{treeof}(S', T')$. Тогда $(A', B') = (A_1 \times A_2, B_1 \times B_2)$, где $A_i = \text{treeof}(S_i)$, а $B_i = \text{treeof}(B_i)$. Поскольку $(A', B') \in \nu S$, из S -консистентности множества νS следует $(A_i, B_i) \in \nu S$, а отсюда $(S_i, T_i) \in R$ по определению R . Теперь определение S_m дает нам $(S', T') = (S_1 \times S_2, T_1 \times T_2) \in S_m(R)$.

Вариант: $(S', T') = (S_1 \rightarrow S_2, T_1 \rightarrow T_2)$

Аналогично.

Вариант: $(S', T') = (S', \mu X. T_1)$

Пусть $T' = [X \mapsto \mu X. T_1] T_1$. По определению, $\text{treeof}(T') = \text{treeof}(T)$. Следовательно, по определению R , мы имеем $(S', T') \in R$, и, таким образом, $(S', T') \in S_m(R)$, по определению S_m .

Вариант: $(S', T') = (\mu X. S_1, T')$

Если $T' = \text{Top}$ или T' начинается с μ , то применим один из предыдущих вариантов; в противном случае, доказательство аналогично предпоследнему варианту.

Соответствие, установленное этой теоремой, является утверждением о корректности и полноте отношения подтипов для μ -типов, определенного в этом разделе. Оно аналогично обыкновенному отношению подтипов для бесконечных древовидных типов, которые ограничены типами, представимыми в виде конечных μ -выражений.

21.9. Подсчет подвыражений

При подстановке в общий алгоритм gfp^t (21.6.4) конкретной функции $support_{S_m}$ для отношения подтипов на μ -типах (21.8.4) получается алгоритм проверки подтипов, изображенный на рис. 21.4. Рассуждения, приведенные в разделе 21.6, показывают, что для этого алгоритма будет гарантировано завершение в случае, если множество $reachable_{S_m}(S, T)$ конечно для любой пары μ -типов (S, T) . Настоящий раздел посвящен доказательству того, что это условие выполняется (утверждение 21.9.11).

$subtype_{S_m}(A, S, T) =$ если $(S, T) \in A$, то
 A
 иначе пусть $A_0 = A \cup \{(S, T)\}$, и тогда
 если $T = \text{Top}$, то
 A_0
 иначе если $S = S_1 \times S_2$ и $T = T_1 \times T_2$, то
 пусть $A_1 = subtype(A_0, S_1, T_1)$, и тогда
 $subtype(A_1, S_2, T_2)$
 иначе если $S = S_1 \rightarrow S_2$ и $T = T_1 \rightarrow T_2$, то
 пусть $A_1 = subtype(A_0, T_1, S_1)$, и тогда
 $subtype(A_1, S_2, T_2)$
 иначе если $T = \mu X. T_1$, то
 $subtype(A_0, S, [X \mapsto \mu X. T_1]T_1)$
 иначе если $S = \mu X. S_1$, то
 $subtype(A_0, [X \mapsto \mu X. S_1]S_1, T)$
 иначе
неудача

Рис. 21.4. Конкретный алгоритм проверки подтипов для μ -типов

На первый взгляд, это свойство кажется почти очевидным, однако его строгое доказательство оказывается удивительно трудоемким. Сложность состоит в том, что есть два способа определения множества «замкнутых подвыражений» μ -типа. Один из них, который мы назовем «подвыражения при взгляде сверху вниз», прямо соответствует подвыражениям, порождаемым через $support_{S_m}$. Другой, который мы назовем «подвыражения при взгляде снизу вверх», приводит к простому доказательству того, что множество замкнутых подвыражений всякого μ -типа конечно. Доказательство завершения алгоритма строится так: сначала определяются оба этих способа, а потом показывается, что множество подвыражений по первому определению является подмножеством множества подвыражений по второму (утверждение 21.9.10). Такой способ доказательства взят нами у Брандта и Хенглейна (Brandt and Henglein, 1997).

Определение 21.9.1 μ -тип S является подвыражением (*top-down subexpression*) μ -типа T при взгляде сверху вниз, что записывается как $S \sqsubseteq T$, если пара (S, T) принадлежит наименьшей неподвижной точке следующей порождающей функции:

$$\begin{aligned}
 TD(R) &= \{(T, T) \mid T \in \mathcal{T}_m\} \\
 &\cup \{(S, T_1 \times T_2) \mid (S, T_1) \in R\} \\
 &\cup \{(S, T_1 \times T_2) \mid (S, T_2) \in R\} \\
 &\cup \{(S, T_1 \rightarrow T_2) \mid (S, T_1) \in R\} \\
 &\cup \{(S, T_1 \rightarrow T_2) \mid (S, T_2) \in R\} \\
 &\cup \{(S, \mu X. T) \mid (S, [X \mapsto \mu X. T]T) \in R\}
 \end{aligned}$$

Упражнение 21.9.2 [★]: Дайте эквивалентное определение отношения $S \sqsubseteq T$ в виде набора правил вывода.

Исходя из определения $support_{S_m}$, нетрудно видеть, что для любых μ -типов S и T все пары, содержащиеся в $support_{S_m}(S, T)$, состоят из подвыражений S и T при взгляде сверху вниз.

Лемма 21.9.3 Если $(S', T') \in support_{S_m}(S, T)$, то либо $S' \sqsubseteq S$, либо $S' \sqsubseteq T$, а также либо $T' \sqsubseteq S$, либо $T' \sqsubseteq T$.

Доказательство: Следует непосредственно из определения $support_{S_m}$.

Кроме того, отношение подвыражений при взгляде сверху вниз транзитивно:

Лемма 21.9.4 Если $S \sqsubseteq U$ и $U \sqsubseteq T$, то $S \sqsubseteq T$.

Доказательство: Утверждение леммы эквивалентно $\forall U, T. U \sqsubseteq T \Rightarrow (\forall S. S \sqsubseteq U \Rightarrow S \sqsubseteq T)$. Другими словами, требуется показать, что $\mu(TD) \subseteq R$, где $R = \{(U, T) \mid \forall S. S \sqsubseteq U \Rightarrow S \sqsubseteq T\}$. По принципу индукции, достаточно показать, что отношение R является TD -замкнутым — то есть, что $TD(R) \subseteq R$. Предположим, что $(U, T) \in TD(R)$. Проведем разбор вариантов по определению TD .

Вариант: $(U, T) = (T, T)$

Ясно, что $(T, T) \in R$.

Вариант: $(U, T) = (U, T_1 \times T_2)$ и $(U, T_1) \in R$

Поскольку $(U, T_1) \in R$, должно быть $S \sqsubseteq U \Rightarrow S \sqsubseteq T_1$ для любого S . По определению \sqsubseteq , для всех S должно выполняться также $S \sqsubseteq U \Rightarrow S \sqsubseteq T_1 \times T_2$. Следовательно, $(U, T) = (U, T_1 \times T_2) \in R$, согласно определению R .

Остальные варианты:

Аналогично.

Сочетание двух последних лемм дает нам утверждение, ради которого мы ввели понятие подвыражения при взгляде сверху вниз:

Утверждение 21.9.5 Если $(S', T') \in reachable_{S_m}(S, T)$, то либо $S' \sqsubseteq S$, либо $S' \sqsubseteq T$, а также либо $T' \sqsubseteq S$, либо $T' \sqsubseteq T$.

Доказательство: Индукция по определению $reachable_{S_m}$, с использованием транзитивности \sqsubseteq .

Конечность $reachable_{S_m}(S, T)$ будет следовать (в утверждении 21.9.11) из последнего доказанного утверждения и из того факта, что у всякого μ -типа U имеется только конечное число подвыражений при взгляде сверху вниз. К сожалению, это последнее утверждение неочевидно из определения \sqsubseteq . Попытка доказать это путем структурной индукции по U , используя определение TD , терпит неудачу, поскольку последний вариант в определении TD ломает индукцию: для построения подвыражений $U = \mu X.T$ используется выражение $[X \mapsto \mu X.T]T$, а оно, возможно, *больше* по размеру.

В альтернативном понятии подвыражений при взгляде снизу вверх эту проблему удается обойти, поскольку подстановка μ -типов вместо переменных рекурсии проводится *после* вычисления подвыражений, а не до него. Такое изменение дает нам простое доказательство конечности.

Определение 21.9.6 μ -тип S является подвыражением (*bottom-up subexpression*) μ -типа T при взгляде снизу вверх, что записывается как $S \leq T$, если пара (S, T) принадлежит к наименьшей неподвижной точке следующей порождающей функции:

$$\begin{aligned} BU(R) &= \{(T, T) \mid T \in \mathcal{T}_m\} \\ &\cup \{(S, T_1 \times T_2) \mid (S, T_1) \in R\} \\ &\cup \{(S, T_1 \times T_2) \mid (S, T_2) \in R\} \\ &\cup \{(S, T_1 \rightarrow T_2) \mid (S, T_1) \in R\} \\ &\cup \{(S, T_1 \rightarrow T_2) \mid (S, T_2) \in R\} \\ &\cup \{([X \mapsto \mu X. T]S, \mu X. T) \mid (S, T) \in R\} \end{aligned}$$

Это новое определение подвыражений отличается от старого только в варианте для типов, начинающихся с символа μ . Чтобы получить подвыражения такого типа при взгляде сверху вниз, мы сначала его разворачивали, а затем собирали подвыражения результата развертки. Вычисляя подвыражения при взгляде снизу вверх, мы сначала собираем (не обязательно замкнутые) подвыражения тела, а затем замыкаем их, применяя развертывающую подстановку.

Упражнение 21.9.7 [★★]: Дайте эквивалентное определение отношения $S \leq T$ в виде набора правил вывода.

Без труда доказывается, что всякое выражение имеет лишь конечное число подвыражений при взгляде снизу вверх.

Лемма 21.9.8 Множество $\{S \mid S \leq T\}$ конечно для любого T .

Доказательство: Прямолинейная структурная индукция по T , используя следующие наблюдения, которые следуют из определений BU и отношения \leq :

- если $T = \text{Top}$ или $T = X$, то $\{S \mid S \leq T\} = \{T\}$;
- если $T = T_1 \times T_2$ либо $T = T_1 \rightarrow T_2$, то $\{S \mid S \leq T\} = \{T\} \cup \{S \mid S \leq T_1\} \cup \{S \mid S \leq T_2\}$;
- если $T = \mu X. T'$, то $\{S \mid S \leq T\} = \{T\} \cup \{[X \mapsto T]S \mid S \leq T'\}$.

Чтобы доказать, что подвыражения типа при взгляде снизу вверх включают его подвыражения при взгляде сверху вниз, нам потребуется следующая лемма, связывающая подвыражения при взгляде снизу вверх и подстановку.

Лемма 21.9.9 Если $S \leq [X \mapsto Q]T$, то либо $S \leq Q$, либо $S = [X \mapsto Q]S'$ для некоторого S' такого, что $S' \leq T$.

Доказательство: Структурная индукция по T .

Вариант: $T = \text{Top}$

В определении BU только вариант с рефлексивностью допускает Top в качестве правого элемента пары, так что должно быть $S = \text{Top}$. Требуемый результат получается при $S' = \text{Top}$.

Вариант: $T = Y$

Если $Y = X$, имеем $S \leq [X \mapsto Q]T = Q$, и требуемый результат следует из условия леммы. Если $Y \neq X$, имеем $S \leq [X \mapsto Q]T = Y$. Только вариант с рефлексивностью в определении BU может дать нам эту пару, так что $S = Y$. Требуемый результат получается при $S' = Y$.

Вариант: $T = T_1 \times T_2$

Имеем $S \leq [X \mapsto Q]T = [X \mapsto Q]T_1 \times [X \mapsto Q]T_2$. Согласно определению BU , есть три способа, которыми S может оказаться подвыражением этого типа-произведения при взгляде снизу вверх. Рассмотрим эти способы по очереди.

Подвариант: $S = [X \mapsto Q]T$

Возьмем $S' = T$.

Подвариант: $S \leq [X \mapsto Q]T_1$

Согласно предположению индукции, либо $S \leq Q$ (и тогда мы уже имеем требуемое утверждение), либо $S = [X \mapsto Q]S'$ для некоторого $S' \leq T_1$. В последнем случае получаем требуемый результат $S' \leq [X \mapsto Q]$ по определению BU .

Подвариант: $S \leq [X \mapsto Q]T_2$

Аналогично.

Вариант: $T = T_1 \rightarrow T_2$

Аналогично варианту с типом-произведением.

Вариант: $T = \mu Y. T'$

Имеем $S \leq [X \mapsto Q]T = \mu Y. [X \mapsto Q]T'$. Есть два способа, которыми S может оказаться подвыражением этого типа при взгляде снизу вверх.

Подвариант: $S = [X \mapsto Q]T$

Возьмем $S' = T$.

Подвариант: $S = [Y \mapsto \mu Y. [X \mapsto Q]T]S_1$, где $S_1 \leq [X \mapsto Q]T'$

Предположение индукции дает нам две возможности:

- $S_1 \leq Q$. Исходя из наших соглашений об именах связанных переменных, известно, что $Y \notin FV(Q)$, и отсюда $Y \notin FV(S_1)$. Но тогда $S = [Y \mapsto \mu Y. [X \mapsto Q]T]S_1 = S_1$, так что $S \leq Q$.
- $S_1 = [X \mapsto Q]S_2$ для некоторого S_2 такого, что $S_2 \leq T'$. В таком случае, $S = [Y \mapsto \mu Y. [X \mapsto Q]T]S_1 = [Y \mapsto \mu Y. [X \mapsto Q]T][X \mapsto Q]S_2 = [X \mapsto Q][Y \mapsto \mu Y. T]S_2$. Требуемый результат получается при $S' = [Y \mapsto \mu Y. S]S_2$.

В заключительной части доказательства мы показываем, что всякое подвыражение μ -типа при взгляде сверху вниз можно найти среди его подвыражений при взгляде снизу вверх.

Утверждение 21.9.10 *Если $S \sqsubseteq T$, то $S \leq T$.*

Доказательство: Требуется показать, что $\mu TD \subseteq \mu BU$. По принципу индукции, достаточно показать, что μBU является TD -замкнутым, то есть, что $TD(\mu BU) \subseteq \mu BU$. Другими словами, нам нужно показать, что из $(A, B) \in TD(\mu BU)$ следует, что $(A, B) \in \mu BU = BU(\mu BU)$. Последнее утверждение будет верно, если для каждого варианта TD , который мог бы породить пару (A, B) из μBU , найдется вариант в определении BU , который также порождал бы (A, B) из μBU . Это trivialно верно для всех вариантов в определении TD , кроме последнего, поскольку они совпадают с соответствующими вариантами BU . В последнем варианте $(A, B) = (S, \mu X. T) \in TD(\mu BU)$ и $(S, [X \mapsto \mu X. T]T) \in \mu BU$, или, что то же самое, $S \leq [X \mapsto \mu X. T]T$. По лемме 21.9.9, либо $S \leq \mu X. T$, то есть, $(S, \mu X. T) \in \mu BU$, что нам и надо, либо $S = [X \mapsto \mu X. T]S'$ для некоторого S' такого, что $(S', T) \in \mu BU$. Из последнего утверждения следует, что $(S, \mu X. T) \in BU(\mu BU) = \mu BU$, по последнему варианту в определении BU .

Комбинируя утверждения, доказанные в этом разделе, мы получаем окончательный результат.

Утверждение 21.9.11 Для любых μ -типов S и T , множество $reachable_{S_m}(S, T)$ конечно.

Доказательство: Пусть для S и T , Td будет множеством их подвыражений при взгляде сверху вниз, а Bu — множеством их подвыражений при взгляде снизу вверх. Согласно утверждению 21.9.5, $reachables_m(S, T) \subseteq Td \times Td$. Согласно утверждению 21.9.10, $Td \times Td \subseteq Bu \times Bu$. По лемме 21.9.8, последнее множество конечно. Следовательно, множество $reachable_{S_m}(S, T)$ также конечно.

21.10. Отступление: экспоненциальный алгоритм

Алгоритм *subtype*, представленный в начале §21.9 (рис. 21.4), можно еще немножко упростить, заставив его возвращать булевское значение, а не новое множество гипотез (см. рис. 21.5). Полученная при этом процедура, $subtype^{ac}$, соответствует алгоритму Амадио и Карделли для проверки подтипов (Amadio and Cardelli, 1993). Она вычисляет то же самое отношение, что и *subtype*, но намного менее эффективна, поскольку не запоминает пары типов в отношении подтипов во время рекурсивных вызовов в вариантах для \rightarrow и \times . Такое, казалось бы, невинное изменение приводит к взрывному росту количества рекурсивных вызовов, выполняемых алгоритмом. Количество рекурсивных вызовов в алгоритме *subtype* пропорционально квадрату общего количества под выражений в двух типах-аргументах (можно это проверить, исследовав доказательства леммы 21.9.8 и утверждения 21.9.11), тогда как в случае *subtype^{ac}* оно растет экспоненциально.

Экспоненциальное поведение *subtype^{ac}* можно ясно показать на следующем примере. Индуктивно определим семейства типов S_n и T_n следующим образом:

$$\begin{array}{lll} S_0 & = & \mu X. \text{Top} \times X \\ T_0 & = & \mu X. \text{Top} \times (\text{Top} \times X) \end{array} \quad \begin{array}{lll} S_{n+1} & = & \mu X. X \rightarrow S_n \\ T_{n+1} & = & \mu X. X \rightarrow T_n \end{array}$$

```

 $\text{subtype}^{ac}(A, S, T) = \begin{cases} \text{если } (S, T) \in A, \text{ то } \text{истина} \\ \text{иначе пусть } A_0 = A \cup (S, T), \text{ и тогда} \\ \quad \text{если } T = \text{Top}, \text{ то } \text{истина} \\ \quad \text{иначе если } S = S_1 \times S_2 \text{ и } T = T_1 \times T_2, \text{ то} \\ \quad \quad \text{ subtype}^{ac}(A_0, S_1, T_1) \text{ и} \\ \quad \quad \text{ subtype}^{ac}(A_0, S_2, T_2) \\ \quad \text{иначе если } S = S_1 \rightarrow S_2 \text{ и } T = T_1 \rightarrow T_2, \text{ то} \\ \quad \quad \text{ subtype}^{ac}(A_0, T_1, S_1) \text{ и} \\ \quad \quad \text{ subtype}^{ac}(A_0, S_2, T_2) \\ \quad \text{иначе если } S = \mu X. S_1, \text{ то} \\ \quad \quad \text{ subtype}^{ac}(A_0, [X \mapsto \mu X. S_1]S_1, T) \\ \quad \text{иначе если } T = \mu X. T_1, \text{ то} \\ \quad \quad \text{ subtype}^{ac}(A_0, S, [X \mapsto \mu X. T_1]T_1) \\ \quad \text{иначе ложь} \end{cases}$ 

```

Рис. 21.5. Алгоритм проверки подтипования Амадио и Карделли

Поскольку S_n и T_n содержат по одному вхождению S_{n-1} и T_{n-1} , соответственно, их размер (после раскрытия сокращений) линейно зависит от n . Однако проверка $S_n <: T_n$ порождает вывод, растущий экспоненциально, как можно видеть по следующей последовательности рекурсивных вызовов

$$\begin{aligned}
& \text{ subtype}^{ac}(\emptyset, S_n, T_n) \\
&= \text{ subtype}^{ac}(A_1, S_n \rightarrow S_{n-1}, T_n) \\
&= \text{ subtype}^{ac}(A_2, S_n \rightarrow S_{n-1}, T_n \rightarrow T_{n-1}) \\
&= \text{ subtype}^{ac}(A_3, T_n, S_n) \text{ и } \underline{\text{ subtype}^{ac}(A_3, S_{n-1}, T_{n-1})} \\
&= \text{ subtype}^{ac}(A_4, T_n \rightarrow T_{n-1}, S_n) \text{ и } \dots \\
&= \text{ subtype}^{ac}(A_5, T_n \rightarrow T_{n-1}, S_n \rightarrow S_{n-1}) \text{ и } \dots \\
&= \text{ subtype}^{ac}(A_6, S_n, T_n) \text{ и } \underline{\text{ subtype}^{ac}(A_6, T_{n-1}, S_{n-1})} \text{ и } \dots \\
&= \text{ и т. д.},
\end{aligned}$$

где

$$\begin{aligned}
A_1 &= \{(S_n, T_n)\} \\
A_2 &= A_1 \cup \{(S_n \rightarrow S_{n-1}, T_n)\} \\
A_3 &= A_2 \cup \{(S_n \rightarrow S_{n-1}, T_n \rightarrow T_{n-1})\} \\
A_4 &= A_3 \cup \{(T_n, S_n)\} \\
A_5 &= A_4 \cup \{(T_n \rightarrow T_{n-1}, S_n)\} \\
A_6 &= A_5 \cup \{(T_n \rightarrow T_{n-1}, S_n \rightarrow S_{n-1})\}
\end{aligned}$$

Обратите внимание, что исходный вызов $\text{ subtype}^{ac}(\emptyset, S_n, T_n)$ приводит к двум (подчеркнутым) рекурсивным вызовам такого же вида, но с аргументами S_{n-1} и T_{n-1} . Каждый из них, в свою очередь, приведет к двум рекурсивным вызовам с аргументами S_{n-2} и T_{n-2} , и так далее. Таким образом, общее количество рекурсивных вызовов пропорционально 2^n .

21.11. Подтипованиe для изорекурсивных типов

В §20.2 мы упомянули, что некоторые системы с рекурсивными типами используют *изорекурсивное* (iso-recursive) представление, в котором свертка и развертка рекурсивных типов явным образом помечаются конструкторами термов `fold` и `unfold`. В таких языках конструктор типов μ является «жестким» в том смысле, что его положение в типе влияет на то, как можно использовать термы данного типа.

Если мы добавим подтипованиe в систему с изорекурсивными типами, жесткость конструктора μ будет влиять также на отношение подтипованиe. Вместо интуитивного понятия «развернуть до предела, а затем проверить подтипованиe», которое мы применяли на протяжении большей части этой главы, необходимо напрямую определить правила подтипованиe для рекурсивных типов.

Наиболее распространенное определение изорекурсивного подтипованиe — так называемое «янтарное правило» (Amber rule), которое называется так, поскольку стало известным по языку Amber («янтарь»), созданному Карделли (Cardelli, 1986):

$$\frac{\Sigma, X <: Y \vdash S <: T}{\Sigma \vdash \mu X . S <: \mu Y . T} \quad (\text{S-AMBER})$$

Интуитивно это правило можно читать так: «Чтобы показать, что $\mu X . S$ является подтипом $\mu Y . T$ при некотором множестве предположений Σ , достаточно показать, что $S <: T$ при дополнительном предположении, что $X <: Y$.⁵ Σ здесь представляет собой просто множество пар переменных рекурсии, где хранятся уже рассмотренные пары рекурсивных типов. Эти предположения используются еще в одном правиле подтипованиe:

$$\frac{(X <: Y) \in \Sigma}{\Sigma \vdash X <: Y} \quad (\text{S-ASSUMPTION})$$

позволяющем нам заключить, что $X <: Y$, если мы уже это предполагаем.

В сущности, добавление этих двух правил к обычному алгоритму проверки подтипованиe из главы 16 (при этом надо еще расширить остальные правила, чтобы они передавали предположения от предпосылок к заключению) дает нам алгоритм, который ведет себя примерно как алгоритм *subtype^{ac}* по рис. 21.5, где Σ играет роль A . Разница состоит в том, что а) «развертка» рекурсивных типов проводится только тогда, когда они оказываются сразу по обе стороны отношения $<:$, и б) не проводится подстановка рекурсивных типов в тела (они просто остаются в виде переменных), и поэтому легко видеть, что алгоритм всегда завершается.

⁵Заметим, что это правило, в отличие от большинства правил, в которых в обеих частях присутствуют связывающие конструкции (например, S-ALL на рис. 26.1), требует, чтобы перед применением правила связанные переменные X и Y были переименованы так, чтобы различаться.

Правила подтипования, встречающиеся в именных системах типов (таких, как Облегченная Java, см. главу 19) близкородственны «янтарному правилу».

Упражнение 21.11.1 [РЕКОМЕНДУЕТСЯ, ★★]: Найдите рекурсивные типы S и T такие, что $S <: T$ по эквирекурсивному определению, но не по «янтарному правилу».

21.12. Дополнительные замечания

Эта глава основана на статье-самоучителе Гапеева, Левина и Пирса (Gapeyev, Levin, and Pierce, 2000).

Основные сведения о коиндукции можно найти в «Порочных кругах» Барвайза и Мосса («Vicious Circles», Barwise and Moss, 1996), самоучителе Гордона по коиндукции и функциональному программированию (Gordon, 1995), а также во вводной статье Милнера и Тофте по коиндукции в семантике языков программирования (Milner and Tofte, 1991a). Информацию о монотонных функциях и неподвижных точках см. у Ачеля (Aczel, 1977) и у Дэви и Пристли (Davey and Priestley, 1990).

Коиндуктивные доказательства стали использоваться в информатике в 1970-е годы, например, в работах Милнера (Milner, 1980) и Парка (Park, 1981) по параллелизму; см. также категориальные рассуждения Арбiba и Мейнса о дуальности в теории автоматов (Arbib and Manes, 1975). Однако индукция в ее двойственной «ко-» форме была известна математикам намного раньше; она в явном виде использовалась, например, в абстрактной алгебре и теории категорий. Основополагающая книга Ачеля (1988) о не вполне обоснованных множествах содержит краткий исторический обзор.

Амадио и Карделли (Amadio and Cardelli, 1993) построили первый алгоритм проверки подтипования для рекурсивных типов. В их статье определяются три отношения: отношение включения на бесконечных деревьях, алгоритм, проверяющий подтиповование для μ -типов, и отношение ссылочно-го подтиповования для μ -типов, определяемое как наименьшая неподвижная точка для набора декларативных правил вывода; доказывается, что эти три отношения эквивалентны, и все они связываются с модельной конструкцией, основанной на отношениях частичной эквивалентности. Коиндукция не используется; вместо этого для рассуждений о бесконечных деревьях вводится понятие конечного приближения к бесконечному дереву. Это понятие затем играет ключевую роль во многих доказательствах.

Брандт и Хенглейн (Brandt and Henglein, 1997) обнажили коиндуктивную суть системы Амадио и Карделли, дав новую индуктивную аксиоматизацию отношения подтипования, корректную и полную по отношению к аксиомам Амадио и Карделли. Так называемое правило ARROW/FIX в этой аксиоматизации воплощает коиндуктивность системы. В этой статье описывается общий метод построения индуктивной аксиоматизации для отношений, которые наиболее естественно описываются через коиндукцию, и приводится подробное доказательство завершения для алгоритма проверки подтипования. В §21.9

мы близко следуем ходу этого доказательства. Брандт и Хенглейн показывают, что сложность их алгоритма — $O(n^2)$.

Козен, Палсберг и Шварцбах ([Kozen, Palsberg, and Schwartzbach, 1993](#)) получают изящный квадратичный алгоритм подтиповирования, замечая, что регулярный рекурсивный тип соответствует автомату с помеченными состояниями. Они определяют понятие произведения для двух автоматов и получают обычновенный автомат над словами, который принимает слово тогда и только тогда, когда типы, соответствующие исходным автоматам, не находятся в отношении подтиповирования. После этого проверка на пустоту, проводимая за линейное время, решает задачу проверки подтиповирования. Это, в сочетании с квадратичным временем для построения автомата-произведения и линейным временем преобразования типов в автоматы, дает общую квадратичную сложность.

Хосояя, Вуйон и Пирс ([Hosoya, Vouillon, and Pierce, 2001](#)) используют родственный подход, основанный на теории автоматов, и устанавливают связь между рекурсивными типами (с объединениями) и древовидными автоматами, что позволяет построить алгоритм проверки подтиповирования, приспособленный для задач обработки XML.

Джим и Палсберг ([Jim and Palsberg, 1999](#)) решают задачу *реконструкции типов* (type reconstruction) (см. гл. 22) для языков с подтипами и рекурсивными типами. Как и мы в этой главе, они используют коиндуктивную точку зрения на отношение подтиповирования для бесконечных деревьев, и обосновывают алгоритм проверки подтиповирования как процедуру, которая строит минимальную симуляцию (т. е., в нашей терминологии, консистентное множество) для данной пары типов. Они определяют понятия консистентности и $P1$ -замыкания для отношения над типами, соответствующие нашим понятиям консистентности и множеств достичимых элементов.

При достаточно долгом
размышлении это утверждение
очевидно.

Сол Горн

Часть V

Полиморфизм

Глава 22

Реконструкция типов

Алгоритмы проверки типов для всех виденных нами до сих пор исчислений зависят от явных аннотаций типа — в частности, им нужно, чтобы в лямбда-абстракциях указывался тип аргумента. В этой главе мы разрабатываем более мощный алгоритм *реконструкции типов* (type reconstruction), способный вычислить *главный тип* (principal type) терма, в котором аннотации полностью или частично отсутствуют. Алгоритмы, родственные нашему, лежат в основе таких языков, как ML или Haskell.

Сочетание реконструкции типов с другими языковыми конструкциями часто требует осторожности. В частности, записи и подтипы представляют заметные трудности. Чтобы упростить изложение, мы в этой книге рассматриваем реконструкцию только для простых типов; в §22.8 можно найти ссылки на более подробные исследования других комбинаций.

22.1. Типовые переменные и подстановки

В некоторых исчислениях из предыдущих глав мы предполагали, что множество типов включает в себя бесконечный набор *неинтерпретируемых* (uninterpreted) базовых типов (§11.1). В отличие от интерпретируемых базовых типов, таких как `Bool` или `Nat`, у этих типов нет никаких операций создания или уничтожения термов; интуитивно, они служат просто заглушками для каких-то конкретных типов, настоящая природа которых нас в данный момент не интересует. В этой главе мы будем задавать вопросы вроде «Если мы заменим заглушку `X` в терме `t` конкретным типом `Bool`, получится ли у нас типизируемый терм?». Другими словами, мы будем рассматривать неинтерпретируемые базовые типы как *типовые переменные* (type variables), которые можно *конкретизировать* (instantiate) с помощью *подстановки* (substitution) другими типами.

С технической точки зрения в этой главе полезно разделить операцию подстановки типов вместо типовых переменных на две части: нужно *описать*

В этой главе рассматривается простое типизированное лямбда-исчисление (рис. 9.1) с булевскими значениями (§1.1), натуральными числами (§8.2) и бесконечным набором базовых типов (§11.1). Соответствующие реализации на OCaml называются `recon` и `fullrecon`.

отображение σ типовых переменных на типы, называемое *подстановкой типов*(type substitution), и применить это отображение к конкретному типу T , получая конкретизацию σT . Например, можно определить $\sigma = [X \mapsto \text{Bool}]$, а затем применить σ к типу $X \rightarrow X$, получая $\sigma(X \rightarrow X) = \text{Bool} \rightarrow \text{Bool}$.

Определение 22.1.1 С формальной точки зрения, подстановка типов (или просто подстановка, если из контекста понятно, что речь идет о типах) представляет собой конечное отображение из типовых переменных в типы. Например, мы записываем в виде $[X \mapsto T, Y \mapsto U]$ подстановку, сопоставляющую T переменной X и U переменной Y . Запись $\text{dom}(\sigma)$ обозначает множество типовых переменных, встречающихся в левой части пар, образующих σ , а $\text{range}(\sigma)$ обозначает множество типов, встречающихся в правой части. Заметим, что одна и та же переменная может принадлежать как области определения, так и области значений подстановки. Как и в случае подстановки термов, предполагается, что все компоненты подстановки применяются одновременно; например, $[X \mapsto \text{Bool}, Y \mapsto X \rightarrow X]$ переводит X в Bool , а Y переводит в $X \rightarrow X$, а не в $\text{Bool} \rightarrow \text{Bool}$.

Применение подстановки к типу определяется очевидным образом:

$$\begin{aligned}\sigma(X) &= \begin{cases} T & \text{если } (X \mapsto T) \in \sigma \\ X & \text{если } X \notin \text{dom}(\sigma) \end{cases} \\ \sigma(\text{Nat}) &= \text{Nat} \\ \sigma(\text{Bool}) &= \text{Bool} \\ \sigma(T_1 \rightarrow T_2) &= \sigma T_1 \rightarrow \sigma T_2\end{aligned}$$

Заметим, что при подстановке типов нет необходимости принимать меры для того, чтобы избежать захвата переменных, поскольку в языке выражений над типами нет конструкций, связывающих переменные (такие конструкции появятся в главе 23).

Подстановка типов поточечно расширяется на контексты при помощи следующего определения:

$$\sigma(x_1 : T_1, \dots, x_n : T_n) = (x_1 : \sigma T_1, \dots, x_n : \sigma T_n)$$

Аналогично, подстановка применяется к терму t путем применения ко всем типам, встречающимся в его аннотациях.

Если имеются подстановки σ и γ , мы обозначим записью $\sigma \circ \gamma$ подстановку, которая получается их композицией по следующим правилам:

$$\sigma \circ \gamma = \left[\begin{array}{ll} X \mapsto \sigma(T) & \text{для всех } (X \mapsto T) \in \gamma \\ X \mapsto T & \text{для всех } (X \mapsto T) \text{ при } X \notin \text{dom}(\gamma) \end{array} \right]$$

Заметим, что $(\sigma \circ \gamma)S = \sigma(\gamma S)$.

Ключевое свойство подстановок типов состоит в том, что они сохраняют корректность утверждений о типизации: если терм, содержащий типовые переменные, правильно типизирован, таков же будет и результат применения к нему произвольной подстановки.

Теорема 22.1.2 [Сохранение типизации при подстановке типов]: Если σ — какая-либо подстановка типов, а $\Gamma \vdash t : T$, то $\sigma\Gamma \vdash \sigma t : \sigma T$.
Доказательство: Прямолинейная индукция по деревьям вывода типов.

22.2. Две точки зрения на типовые переменные

Предположим, у нас есть терм t , содержащий типовые переменные, и связанный с ним контекст Γ (который также может содержать типовые переменные). Можно задать два существенно различных вопроса, касающихся t :

1. «*Все ли конкретизации t правильно типизированы?*» То есть, верно ли, что при любой σ мы будем иметь $\sigma\Gamma \vdash \sigma t : T$ для некоторого T ?
2. «*Имеются ли правильно типизированные конкретизации t ?*» То есть, можем ли мы найти такую σ , что $\sigma\Gamma \vdash \sigma t : T$ для некоторого T ?

Согласно первой точке зрения, типы переменных должны *оставаться абстрактными* во время проверки типов; таким образом, мы можем быть уверены, что правильно типизированный терм будет правильно себя вести, какие бы типы мы ни подставили вместо его типовых переменных. Например, терм

$\lambda f : X \rightarrow X. \lambda a : X. f (f a);$

имеет тип $(X \rightarrow X) \rightarrow X \rightarrow X$ и, каким бы конкретным типом T мы ни заменили бы X , конкретизация

$\lambda f : T \rightarrow T. \lambda a : T. f (f a);$

окажется правильно типизирована. Отношение к типовым переменным как к абстракциям типа приводит к *параметрическому полиморфизму* (parametric polymorphism), при котором типовые переменные используются для передачи того факта, что терм можно использовать в различных контекстах с различными конкретными типами. Мы вернемся к параметрическому полиморфизму далее в этой главе (§22.7), а также, более подробно, в главе 23.

Со второй точки зрения, терм t даже не обязан быть правильно типизирован; мы хотим узнать, возможно ли *конкретизировать* его до правильно типизированного терма, выбрав подходящие значения для каких-либо из его типовых переменных. Например, терм

$\lambda f : Y. \lambda a : X. f (f a);$

в приведенном виде не типизируем, однако, если мы заменим Y на $\text{Nat} \rightarrow \text{Nat}$, а X — на Nat , мы получим терм

$\lambda f : \text{Nat} \rightarrow \text{Nat}. \lambda a : \text{Nat}. f (f a);$

типа $(\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat}$. Или же, при простой замене Y на $X \rightarrow X$, мы получим терм

$\lambda f : X \rightarrow X. \lambda a : X. f (f a);$

правильно типизированный, несмотря на наличие переменных. На самом деле, этот терм является *наиболее общей* (most general) конкретизацией $\lambda f : Y. \lambda a : X. f (f a)$ в том смысле, что такая подстановка принимает наименьшие обязательства относительно значений типовых переменных, среди всех подстановок, дающих в результате правильно типизированный терм.

Поиск корректных конкретизаций для типовых переменных является основой идеи *реконструкции типов* (type reconstruction) (иногда ее еще называют

выводом типов, type inference), в рамках которой компилятор помогает восстановить информацию о типах, опущенную программистом. В предельном случае можно, как в ML, позволить программисту опускать *все* аннотации типа и писать на языке чистого, бестипового лямбда-исчисления. Во время синтаксического анализа мы снабжаем каждую «голую» лямбда-абстракцию $\lambda x.t$ переменной типа, получая $\lambda x:X.t$. При этом X выбирается так, чтобы отличаться от типовых переменных всех других абстракций в программе. Затем мы проводим реконструкцию типов и находим самые общие значения для всех этих переменных, позволяющие терму пройти проверку типов. (Схема несколько усложняется, если в языке, как в ML, присутствует полиморфизм через `let`; мы вернемся к этому вопросу в §22.6 и §22.7.)

Чтобы формализовать реконструкцию типов, нам потребуется способ кратко обозначить возможные способы подстановки типов вместо типовых переменных в терме и связанном с ним контексте, при которых получается правильное утверждение о типизации.¹

Определение 22.2.1 Пусть имеется контекст Γ и терм t . Решением (*solution*) для (Γ, t) называется пара (σ, T) такая, что $\sigma\Gamma \vdash \sigma t : T$.

Пример 22.2.2 Пусть $\Gamma = f:X, a:Y, a\ t = f\ a$. Тогда

$$\begin{array}{ll} ([X \mapsto Y \rightarrow \text{Nat}], \text{Nat}) & ([X \mapsto Y \rightarrow Z], Z) \\ ([X \mapsto Y \rightarrow Z, Z \mapsto \text{Nat}], Z) & ([X \mapsto Y \rightarrow \text{Nat} \rightarrow \text{Nat}], \text{Nat} \rightarrow \text{Nat}) \\ ([X \mapsto \text{Nat} \rightarrow \text{Nat}, Y \mapsto \text{Nat}], \text{Nat}) & \end{array}$$

суть различные решения (Γ, t) .

Упражнение 22.2.3 [$\star \rightarrow$]: Найдите три различных решения для терма

$$\lambda x:X. \ \lambda y:Y. \ \lambda z:Z. \ (x\ z) \ (y\ z)$$

в пустом контексте.

22.3. Типизация на основе ограничений

Опишем алгоритм, который принимает на вход терм t и контекст Γ , и вычисляет набор ограничений, т. е. уравнений из выражений типа (возможно, содержащих типовые переменные), которым должны удовлетворять все решения для (Γ, t) . Интуитивная идея, лежащая в основе этого алгоритма, в сущности, та же, что и в основе алгоритма проверки типов; однако новый алгоритм не *роверяет* ограничения, а просто *выписывает* их для дальнейшего рассмотрения. Например, если алгоритм получает терм-применение $t_1\ t_2$, где $\Gamma \vdash t_1 : T_1$ и $\Gamma \vdash t_2 : T_2$, то вместо того, чтобы проверить, что T_1 имеет

¹Эти определения можно организовать иначе. Прежде всего, можно использовать механизм *экзистенциальных унификиандов* (existential unificands), предложенный Кирхнером и Жуанно (Kirchner and Jouannaud, 1990), вместо отдельных условий новизны в правилах порождения ограничений на рис. 22.1. Еще одно возможное улучшение, которое можно найти у Реми (Rémy, 1992a,b, полная версия Rémy, 1998, гл. 5), — рассмотрение самих утверждений о типизации как унификиандов; начиная с тройки (Γ, t, T) , в которой все три компонента могут содержать типовые переменные, мы ищем подстановки σ такие, что $\sigma\Gamma \vdash \sigma(t) : \sigma(T)$, т. е. подстановки, унифицирующие схему утверждения о типизации $\Gamma \vdash t : T$.

$\frac{x:T \in \Gamma}{\Gamma \vdash x : T _{\emptyset} \{ \}} \quad (\text{CT-VAR})$ $\frac{\Gamma, x:T_1 \vdash t_2 : T_2 _{\mathcal{X}} C}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2 _{\mathcal{X}} C} \quad (\text{CT-ABS})$ $\frac{\begin{array}{c} \Gamma \vdash t_1 : T_1 _{\mathcal{X}_1} C_1 \\ \Gamma \vdash t_2 : T_2 _{\mathcal{X}_2} C_2 \\ \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(T_2) = \mathcal{X}_2 \cap FV(T_1) \\ x \notin \mathcal{X}_1, \mathcal{X}_2, T_1, T_2, C_1, C_2, \Gamma, t_1 \text{ или } t_2 \\ C' = C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow x\} \end{array}}{\Gamma \vdash t_1 t_2 : x _{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{x\}} C'} \quad (\text{CT-APP})$ $\Gamma \vdash 0 : \text{Nat} _{\emptyset} \{} \quad (\text{CT-ZERO})$	$\frac{\begin{array}{c} \Gamma \vdash t_1 : T _{\mathcal{X}} C \\ C' = C \cup \{T = \text{Nat}\} \end{array}}{\Gamma \vdash \text{pred } t_1 : \text{Nat} _{\mathcal{X}} C'} \quad (\text{CT-PRED})$ $\frac{\begin{array}{c} \Gamma \vdash t_1 : T _{\mathcal{X}} C \\ C' = C \cup \{T = \text{Nat}\} \end{array}}{\Gamma \vdash \text{iszero } t_1 : \text{Bool} _{\mathcal{X}} C'} \quad (\text{CT-ISZERO})$ $\Gamma \vdash \text{true} : \text{Bool} _{\emptyset} \{} \quad (\text{CT-TRUE})$ $\Gamma \vdash \text{false} : \text{Bool} _{\emptyset} \{} \quad (\text{CT-FALSE})$
$\frac{\begin{array}{c} \Gamma \vdash t_1 : T _{\mathcal{X}} C \\ C' = C \cup \{T = \text{Nat}\} \end{array}}{\Gamma \vdash \text{succ } t_1 : \text{Nat} _{\mathcal{X}} C'} \quad (\text{CT-SUCC})$	$\frac{\begin{array}{c} \Gamma \vdash t_1 : T_1 _{\mathcal{X}_1} C_1 \\ \Gamma \vdash t_2 : T_2 _{\mathcal{X}_2} C_2 \\ \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \text{ не пересекаются} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Bool}, T_2 = T_3\} \end{array}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 _{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3} C'} \quad (\text{CT-IF})$

Рис. 22.1. Правила типизации с ограничениями

вид $T_2 \rightarrow T_{12}$, и вернуть T_{12} в качестве типа целого терма, он выбирает новую типовую переменную X , выписывает ограничение $T_1 = T_2 \rightarrow X$, и возвращает X в качестве типа терма-применения.

Определение 22.3.1 Множество ограничений (*constraint set*) C — это набор уравнений $\{S_i = T_i^{i \in 1..n}\}$. Подстановка σ унифицирует (*unifies*) уравнение $S = T$, если результаты подстановки σS и σT совпадают. Мы говорим, что подстановка σ унифицирует C (или удовлетворяет ему), если она унифицирует все уравнения в C .

Определение 22.3.2 Отношение типизации с ограничениями (*constraint typing relation*) $\Gamma \vdash t : T |_{\mathcal{X}} C$ определяется правилами, представленными на рис. 22.1. Выражение $\Gamma \vdash t : T |_{\mathcal{X}} C$ можно неформально прочитать как «терм t имеет тип T в контексте типизации Γ всегда, когда выполнены ограничения C ». В правиле Т-АРР запись $FV(T)$ обозначает множество всех типовых переменных, упомянутых в T .

Индексы \mathcal{X} помогают отслеживать типовые переменные, введенные в каждом подвыходе. С их помощью мы добиваемся того, чтобы новые переменные,

введенные в разных поддеревьях, были всегда различны. При первом прочтении правил можно не обращать внимания на эти индексы и на предпосылки, в которых они упоминаются. При следующем чтении обратите внимание, что аннотации и предпосылки обеспечивают два условия. Во-первых, каждый раз, когда в последнем правиле некоторого вывода выбирается имя типовой переменной, это имя должно отличаться от имен, упоминающихся в каком-либо из подвыводов. Во-вторых, когда в правиле упомянуты два или более подвыводов, множества переменных, используемых в этих подвыводах, не должны пересекаться. Заметим также, что эти ограничения не запрещают нам строить *какой-нибудь* вывод данного терма; они только запрещают нам строить выводы, в которых одна и та же «новая» переменная вводится в нескольких местах. Поскольку запас имен типовых переменных бесконечен, нам всегда удастся выполнить требование новизны.

При чтении снизу вверх правила типизации с ограничениями определяют несложную процедуру, которая, принимая на входе Γ и t , вычисляет T и C (а также X), такие, что $\Gamma \vdash t : T |_X C$. Однако в отличие от обыкновенного алгоритма типизации для простого типизированного лямбда-исчисления, новый алгоритм никогда не терпит неудачу: для любых Γ и t всегда имеются какие-либо T и C , так что $\Gamma \vdash t : T |_X C$. Более того, T и C однозначно определяются при заданных Γ и t . (Строго говоря, алгоритм является детерминистским, только если его рассматривать «с точностью до выбора новых имен типа». Мы вернемся к этому вопросу в упражнении 22.3.9.)

Чтобы упростить нотацию в последующих рассуждениях, мы иногда опускаем X и пишем просто $\Gamma \vdash t : T | C$.

Упражнение 22.3.3 [★ →]: Постройте вывод типизации с ограничениями, имеющий заключение

$$\vdash \lambda x:X. \lambda y:Y. \lambda z:Z. (x z) (y z) : S |_X C$$

для некоторых S , X и C .

Идея отношения типизации с ограничениями состоит в том, что мы можем проверить, типизируем ли терм t в контексте Γ , следующим образом: сначала собираем ограничения C , которым нужно удовлетворить, чтобы t имел тип, и одновременно получаем тип S , содержащий переменные из C и характеризующий возможные типы t с точностью до этих переменных. Затем, чтобы найти решения для t , мы просто ищем подстановки σ , удовлетворяющие ограничениям C (т. е. превращающие все уравнения из C в тождество); для каждой такой σ , тип σS есть возможный тип t . Если мы обнаружим, что *нет* подстановок, удовлетворяющих ограничениям C , значит, t невозможно конкретизировать так, чтобы он стал типизируемым.

Например, множество ограничений, порождаемых алгоритмом для терма $t = \lambda x:X \rightarrow Y. x 0$, равно $\{\text{Nat} \rightarrow Z = X \rightarrow Y\}$, а связанный с этим множеством тип результата — $(X \rightarrow Y) \rightarrow Z$. Подстановка $\sigma = [X \mapsto \text{Nat}, Z \mapsto \text{Bool}, Y \mapsto \text{Bool}]$ превращает уравнение $\text{Nat} \rightarrow Z = X \rightarrow Y$ в тождество, и, таким образом, мы видим, что $\sigma((X \rightarrow Y) \rightarrow Z)$, т. е. $(\text{Nat} \rightarrow \text{Bool}) \rightarrow \text{Bool}$, является возможным типом для t .

Эта идея формально выражена в следующем определении:

Определение 22.3.4 Допустим, $\Gamma \vdash t : S \mid C$. Решением (*solution*) для (Γ, t, S, C) называется пара (σ, T) такая, что σ удовлетворяет ограничениям C , а $\sigma S = T$.

Алгоритмическая задача поиска подстановок, унифицирующих данный набор ограничений C , будет рассмотрена в следующем разделе. Однако сначала требуется проверить, что наш алгоритм типизации с ограничениями соответствует изначальному декларативному отношению типизации.

При данных контексте Γ и терме t у нас есть два способа охарактеризовать возможные способы конкретизации типовых переменных в Γ и t , дающие в результате корректную типизацию:

1. [Декларативный] как множество всех решений для (Γ, t) в смысле определения 22.2.1; либо
2. [Алгоритмический] через отношение типизации с ограничениями — нужно найти S и C такие, что $\Gamma \vdash t : S \mid C$, а затем взять множество решений (Γ, t, S, C) .

Мы доказываем эквивалентность этих характеризаций в два шага. Сначала покажем, что всякое решение (Γ, t, S, C) является также решением (Γ, t) (теорема 22.3.5). Затем покажем, что всякое решение для (Γ, t) можно *расширить* до решения (Γ, t, S, C) путем присваивания значений типовым переменным, введенным в процессе порождения ограничений.

Теорема 22.3.5 [Корректность типизации с ограничениями]: Предположим, что $\Gamma \vdash t : S \mid C$. Если (σ, T) является решением (Γ, t, S, C) , то оно также является решением (Γ, t) .

Для этого направления рассуждений множества новых переменных X несущественны, и их можно опустить.

Доказательство: Индукция по данному нам выводу типизации с ограничениями $\Gamma \vdash t : S \mid C$. Рассматриваем варианты последнего примененного правила.

Вариант CT-VAR: $t = x \quad x:S \in \Gamma \quad C = \{\}$

Дано, что (σ, T) является решением (Γ, t, S, C) ; поскольку C пусто, это просто означает, что $\sigma S = T$. Но тогда по правилу T-VAR мы немедленно получаем $\sigma\Gamma \vdash x : T$, что и требуется.

Вариант CT-ABS: $t = \lambda x:T_1 . t_2 \quad S = T_1 \rightarrow S_2 \quad \Gamma, x:T_1 \vdash t_2 : S_2 \mid C$

Дано, что (σ, T) является решением для (Γ, t, S, C) , т. е. σ унифицирует C , а $T = \sigma S = \sigma T_1 \rightarrow \sigma S_2$. Тогда $(\sigma, \sigma S_2)$ является решением $((\Gamma, x:T_1), t_2, S_2, C)$. Согласно предположению индукции, $(\sigma, \sigma S_2)$ является решением $((\Gamma, x:T_1), t_2)$, т. е., $\sigma\Gamma, x:\sigma T_1 \vdash \sigma t_2 : \sigma S_2$. По правилу T-ABS, $\sigma\Gamma \vdash \lambda x:\sigma T_1 . \sigma t_2 : \sigma T_1 \rightarrow \sigma S_2 = \sigma(T_1 \rightarrow S_2) = T$, как нам и требуется.

Вариант CT-APP: $t = t_1 \quad t_2 \quad S = X$

$\Gamma \vdash t_1 : S_1 \mid C_1 \quad \Gamma \vdash t_2 : S_2 \mid C_2$

$C = C_1 \cup C_2 \cup \{S_1 = S_2 \rightarrow X\}$

По определению, σ унифицирует C_1 и C_2 , а $\sigma S_1 = \sigma(S_2 \rightarrow X)$. Таким образом,

$(\sigma, \sigma S_1)$ и $(\sigma, \sigma S_2)$ являются решениями для (Γ, t_1, S_1, C_1) и (Γ, t_2, S_2, C_2) . Отсюда по предположению индукции имеем $\sigma\Gamma \vdash \sigma t_1 : \sigma S_1$ и $\sigma\Gamma \vdash \sigma t_2 : \sigma S_2$. Однако, поскольку $\sigma S_1 = \sigma S_2 \rightarrow \sigma X$, мы получаем $\sigma\Gamma \vdash \sigma t_1 : \sigma S_2 \rightarrow \sigma X$ и, по правилу T-APP, $\sigma\Gamma \vdash \sigma(t_1 \ t_2) : \sigma X = T$.

Остальные варианты:

Аналогично.

Доказательство *полноты* типизации с ограничениями по отношению к обыкновенному отношению типизации несколько сложнее, поскольку приходится тщательно отслеживать новые имена типов.

Определение 22.3.6 Запись $\sigma \setminus \mathcal{X}$ обозначает подстановку, которая не определена для переменных из \mathcal{X} , а в остальном ведет себя как σ .

Теорема 22.3.7 [Полнота типизации с ограничениями]: Допустим, $\Gamma \vdash t : S |_{\mathcal{X}} C$. Если (σ, T) является решением (Γ, t) , и $\text{dom}(\sigma) \cap \mathcal{X} = \emptyset$, то существует решение (σ', T) для (Γ, t, S, C) такое, что $\sigma' \setminus \mathcal{X} = \sigma$.

Доказательство: Индукция по данному дереву вывода типизации с ограничениями.

Вариант CT-VAR: $t = x \quad x:S \in \Gamma$

В предположении, что (σ, T) является решением (Γ, x) , лемма об инверсии отношения типизации (9.3.1) говорит, что $T = \sigma S$. Но тогда (σ, T) является также решением $(\Gamma, x, S, \{\})$.

Вариант CT-ABS: $t = \lambda x: T_1 . t_2 \quad \Gamma, x: T_1 \vdash t_2 : S_2 |_{\mathcal{X}} C \quad S = T_1 \rightarrow S_2$

В предположении, что (σ, T) является решением $(\Gamma, \lambda x: T_1 . t_2)$, лемма об инверсии отношения типизации дает $\sigma\Gamma, x: \sigma T_1 \vdash \sigma t_2 : T_2$ и $T = \sigma T_1 \rightarrow T_2$ для некоторого T_2 . Согласно предположению индукции, существует решение (σ', T_2) для $((\Gamma, x: T_1), t_2, S_2, C)$, такое, что подстановка $\sigma' \setminus \mathcal{X}$ согласована с σ . При этом \mathcal{X} не может содержать никакие типовые переменные из T_1 . Поэтому $\sigma' T_1 = \sigma T_1$, а $\sigma' S = \sigma'(T_1 \rightarrow S_2) = \sigma T_1 \rightarrow \sigma' S_2 = \sigma T_1 \rightarrow T_2 = T$. Таким образом, мы видим, что (σ', T) является решением $(\Gamma, (\lambda x: T_1 . t_2), T_1 \rightarrow S_2, C)$.

Вариант CT-APP: $t = t_1 \ t_2 \quad \Gamma \vdash t_1 : S_1 |_{\mathcal{X}_1} C_1 \quad \Gamma \vdash t_2 : S_2 |_{\mathcal{X}_2} C_2$

$$\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$$

$$\mathcal{X}_1 \cap FV(S_2) = \emptyset$$

$$\mathcal{X}_2 \cap FV(S_1) = \emptyset$$

X не встречается в $\mathcal{X}_1, \mathcal{X}_2, S_1, S_2, C_1, C_2$

$$S = X \quad \mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\} \quad C = C_1 \cup C_2 \cup \{S_1 = S_2 \rightarrow X\}$$

В предположении, что (σ, T) является решением $(\Gamma, t_1 \ t_2)$, лемма об инверсии отношения типизации дает $\sigma\Gamma \vdash \sigma t_1 : T_1 \rightarrow T$ и $\sigma\Gamma \vdash \sigma t_2 : T_2$.

Согласно предположению индукции, существуют решения $(\sigma_1, T_1 \rightarrow T)$

для (Γ, t_1, S_1, C_1) и (σ_2, T_2) для (Γ, t_2, S_2, C_2) , причем $\sigma_1 \setminus \mathcal{X}_1 = \sigma = \sigma_2 \setminus \mathcal{X}_2$.

Требуется продемонстрировать подстановку σ' , такую, что: (1) $\sigma' \setminus \mathcal{X}$ согласована с σ ; (2) $\sigma' X = T$; (3) σ' унифицирует C_1 и C_2 , и (4) σ' унифицирует

$\{S_1 = S_2 \rightarrow X\}$, т. е., $\sigma' S_1 = \sigma' S_2 \rightarrow \sigma' X$. Мы определяем σ' так:

$$\sigma' = \begin{cases} Y \mapsto U & \text{если } Y \notin \mathcal{X} \text{ и } (Y \mapsto U) \in \sigma \\ Y_1 \mapsto U_1 & \text{если } Y_1 \in \mathcal{X}_1 \text{ и } (Y_1 \mapsto U_1) \in \sigma_1 \\ Y_2 \mapsto U_2 & \text{если } Y_2 \in \mathcal{X}_2 \text{ и } (Y_2 \mapsto U_2) \in \sigma_2 \\ X \mapsto T & \end{cases}$$

Условия (1) и (2) выполняются очевидным образом. Условие (3) выполняется, поскольку \mathcal{X}_1 и \mathcal{X}_2 не пересекаются. Для проверки условия (4) заметим сначала, что дополнительные условия новизны типовых переменных гарантируют нам $FV(S_1) \cap (\mathcal{X}_2 \cup \{X\}) = \emptyset$, так что $\sigma' S_1 = \sigma_1 S_1$. Теперь проведем такое преобразование: $\sigma' S_1 = \sigma_1 S_1 = T_1 \rightarrow T = \sigma_2 S_2 \rightarrow T = \sigma' S_2 \rightarrow \sigma' X = \sigma'(S_2 \rightarrow X)$.

Остальные варианты:

Аналогично.

Следствие 22.3.8 Допустим, $\Gamma \vdash t : S \mid C$. Решение для (Γ, t) существует тогда и только тогда, когда существует решение для (Γ, t, S, C) .

Доказательство: По теоремам 22.3.5 и 22.3.7.

Упражнение 22.3.9 [РЕКОМЕНДУЕТСЯ, ★★]: В промышленном компиляторе, как правило, вместо недетерминистского выбора новой типовой переменной в правиле СТ-АРР применяется функция, порождающая новую типовую переменную при каждом своем вызове, отличную от всех остальных переменных, порождаемых той же функцией при других вызовах, а также от всех типовых переменных, явно упомянутых в контексте проверяемого типа. Поскольку такие глобальные операции «*gensym*» работают через побочные эффекты и скрытую глобальную переменную, о них трудно рассуждать формально. Однако, их поведение можно сымитировать достаточно точно, и при этом достаточно удобно с математической точки зрения, если «пробросить» последовательность неиспользованных имен переменных сквозь правила порождения ограничений.

Пусть F обозначает последовательность неиспользованных имен типовых переменных. Тогда, вместо того, чтобы записывать отношение порождения ограничений как $\Gamma \vdash t : T \mid_C$, мы его записываем как $\Gamma \vdash_F t : T \mid_{F'} C$, где Γ , F и t служат входными данными алгоритма, а T , F' и C – результатами его работы. Каждый раз, когда алгоритму требуется новая переменная, он берет первый элемент F и возврашает остаток как F' .

Запишите правила для этого алгоритма. Докажите, что они эквивалентны, в соответствующем смысле, исходным правилам порождения ограничений.

Упражнение 22.3.10 [РЕКОМЕНДУЕТСЯ, ★★]: Реализуйте алгоритм из упражнения 22.3.9 на ML. Используйте тип данных

```
type ty =
  TyBool
  | TyArr of ty * ty
  | TyId of string
  | TyNat
```

для типов и

```
type constr = (ty * ty) list
```

для множеств ограничений. Кроме того, потребуется представление для бесконечных последовательностей новых имен переменных. Есть множество способов организовать такие последовательности; вот достаточно простой способ, использующий рекурсивный тип данных:

```
type nextuvar = NextUVar of string * uvargenerator
and uvargenerator = unit → nextuvar
```

```
let uvargen =
  let rec f n () = NextUVar(?X_ ^ string_of_int n, f (n+1))
  in f 0
```

То есть, если вызвать функцию *uvargen* с аргументом (), то она возвращает значение вида *NextUVar(x, f)*, где *x* — новое имя переменной, а *f* — еще одна функция такого же вида.

Упражнение 22.3.11 [★★]: Покажите, как можно расширить алгоритм порождения ограничений, чтобы он мог работать с определениями рекурсивных функций общего вида (§11.11).

22.4. Унификация

При поиске решений для множеств ограничений мы используем идею, высказанную Хиндли (Hindley, 1969) и Милнером (Milner, 1978), которая состоит в использовании *унификации* (unification) (Robinson, 1971) для проверки непустоты множества решений, и, если это так, нахождения «наилучшего» из его элементов в том смысле, что все остальные решения могут быть непосредственно получены из него.

Определение 22.4.1 Подстановка σ называется менее конкретной (*less specific*) (или более обобщенной (*more general*)), чем подстановка σ' , если $\sigma' = \gamma \circ \sigma$ для некоторой подстановки γ . Это отношение записывается так: $\sigma \sqsubseteq \sigma'$,

Определение 22.4.2 Главным унифициатором (*principal unifier*) (иногда его называют наиболее общим унифициатором (*most general unifier*)) для множества ограничений C называется подстановка σ , удовлетворяющая C , если при этом $\sigma \sqsubseteq \sigma'$ для всякой подстановки σ' , удовлетворяющей C .

Упражнение 22.4.3 [★]: Выпишите главные унифициаторы (там, где они существуют) для следующих множеств ограничений:

$$\begin{array}{ll} \{X = Nat, Y = X \rightarrow X\} & \{Nat \rightarrow Nat = X \rightarrow Y\} \\ \{X \rightarrow Y = Y \rightarrow Z, Z = U \rightarrow W\} & \{Nat = Nat \rightarrow Y\} \\ \{Y = Nat \rightarrow Y\} & \{\} \text{ (пустое множество ограничений)} \end{array}$$

```

 $\text{unify}(C) = \begin{cases} \text{если } C = \emptyset, \text{ то } [] \\ \text{иначе пусть } \{S = T\} \cup C' = C, \text{ и тогда} \\ \quad \text{если } S = T \\ \quad \quad \text{тогда } \text{unify}(C') \\ \quad \quad \text{иначе если } S = X \text{ и } X \notin FV(T) \\ \quad \quad \quad \text{тогда } \text{unify}([X \mapsto T]C') \circ [X \mapsto T] \\ \quad \quad \text{иначе если } T = X \text{ и } X \notin FV(S) \\ \quad \quad \quad \text{тогда } \text{unify}([X \mapsto S]C') \circ [X \mapsto S] \\ \quad \quad \text{иначе если } S = S_1 \rightarrow S_2 \text{ и } T = T_1 \rightarrow T_2 \\ \quad \quad \quad \text{тогда } \text{unify}(C' \cup \{S_1 = T_1, S_2 = T_2\}) \\ \quad \quad \text{иначе} \\ \quad \quad \quad \text{неудача} \end{cases}$ 

```

Рис. 22.2. Алгоритм унификации

Определение 22.4.4 Алгоритм унификации (*unification algorithm*) для типов приведен на рис. 22.2.² Выражение «пусть $\{S = T\} \cup C' = C$ » во второй строке должно читаться как «выберем ограничение $S = T$ из набора ограничений C и обозначим множество оставшихся ограничений в C символом C' ».

Дополнительные условия $X \notin FV(T)$ в пятой строке и $X \notin FV(S)$ в седьмой известны как *проверка на вхождение* (occur check). Они не дают алгоритму породить решение, содержащее циклическую подстановку вроде $X \mapsto X \rightarrow X$, которая не имеет смысла, если речь идет о конечных выражениях типа. (Если мы расширим наш язык и включим в него бесконечные выражения типа — т. е., рекурсивные типы, как они описаны в главах 20 и 21, — проверку на вхождение можно убрать.)

Теорема 22.4.5 Алгоритм *unify* всегда завершается. При этом он терпит неудачу, если получает на входе невыполнимый набор ограничений, а в противном случае возвращает главный унификатор. Более формально:

1. *unify*(C) завершается для любого C , либо неудачей, либо давая в результате подстановку;
2. если *unify*(C) = σ , то σ — унификатор для C ;
3. если δ — унификатор для C , то *unify*(C) = σ , причем $\sigma \sqsubseteq \delta$.

Доказательство: для доказательства пункта (1), определим степень множества ограничений C как пару (m, n) , где m — число различных типовых переменных в C , а n — общий размер всех типов в C . Несложно проверить, что всякая ветвь алгоритма *unify* либо немедленно завершается (успешно в первой ветви и неудачно в последней), либо вызывает *unify* рекурсивно,

²Заметим, что ничто в этом алгоритме не использует тот факт, что мы унифицируем выражения типов, а не какие-либо другие выражения; тот же алгоритм можно использовать при решении наборов ограничений для любых выражений (первого порядка).

передавая ей новое множество ограничений, степень которого лексикографически меньше степени исходного.

Часть (2) представляет собой прямолинейную индукцию по числу рекурсивных вызовов при вычислении $\text{unify}(C)$. Все варианты тривиальны, за исключением двух ветвей, работающих с переменными. В этих ветвях для доказательства используется тот факт, что если σ унифицирует $[X \mapsto T]D$, то $\sigma \circ [X \mapsto D]$ унифицирует $\{X = T\} \cup D$ для любого множества ограничений D .

В части (3) доказательство снова ведется индукцией по числу рекурсивных вызовов при вычислении $\text{unify}(C)$. Если C — пустое множество, unify немедленно возвращает тривильную подстановку $[]$; поскольку $\delta = \delta \circ []$, мы имеем $[] \sqsubseteq \delta$, как и требуется. Если C непусто, unify выбирает некоторую пару (S, T) из C , и рассматривает варианты форм S и T .

Вариант: $S = T$

Так как подстановка δ является унификатором для C , она унифицирует также и C' . По предположению индукции, $\text{unify}(C) = \sigma$, причем $\sigma \sqsubseteq \delta$, что и требуется.

Вариант: $S = X$ и $X \notin FV(T)$

Поскольку δ унифицирует S и T , имеем $\delta(S) = \delta(T)$. Таким образом, для любого типа U выполняется $\delta(U) = \delta([X \mapsto T]U)$; в частности, поскольку δ унифицирует C' , она также должна унифицировать $[X \mapsto T]C'$. Предположение индукции дает нам $\text{unify}([X \mapsto T]C') = \sigma'$, причем $\delta = \gamma \circ \sigma'$ для некоторой γ . Поскольку $\text{unify}(C) = \sigma' \circ [X \mapsto T]$, достаточно показать, что $\delta = \gamma \circ (\sigma' \circ [X \mapsto T])$. Рассмотрим любую переменную Y . Если $Y \neq X$, очевидно, имеем $(\gamma \circ (\sigma' \circ [X \mapsto T]))Y = (\gamma \circ \sigma')Y = \delta Y$. С другой стороны, как мы уже видели, $(\gamma \circ (\sigma' \circ [X \mapsto T]))X = (\gamma \circ \sigma')T = \delta X$. Сочетая эти два наблюдения, получаем $\delta Y = (\gamma \circ (\sigma' \circ [X \mapsto T]))Y$ для всех переменных Y , т. е., $\delta = (\gamma \circ (\sigma' \circ [X \mapsto T]))$.

Вариант: $T = X$ и $X \notin FV(S)$

Аналогично.

Вариант: $S = S_1 \rightarrow S_2$ и $T = T_1 \rightarrow T_2$

Не представляет труда. Достаточно заметить, что δ является унификатором $\{S_1 \rightarrow S_2 = T_1 \rightarrow T_2\} \cup C'$ тогда и только тогда, когда она унифицирует $C' \cup \{S_1 = T_1, S_2 = T_2\}$.

Если ни одно из этих условий не применимо к S и T , то $\text{unify}(C)$ терпит неудачу. Но это может произойти только в двух случаях: либо S равен Nat , а T — функциональный тип (или наоборот), либо $S = X$ и $X \in T$ (или наоборот). Первый случай явно противоречит предположению, что C унифицируем. Чтобы увидеть, что второй случай также ему противоречит, заметим, что, по этому предположению, $\delta S = \delta T$; если X встречается в T , то δT всегда будет строго больше, чем δS . Таким образом, если $\text{unify}(C)$ терпит неудачу, то множество ограничений C не унифицируемо, что противоречит нашему предположению о том, что δ является для него унификатором; так что этот случай не может возникнуть.

Упражнение 22.4.6 [РЕКОМЕНДУЕТСЯ, ★★]: Реализуйте алгоритм унификации.

22.5. Главные типы

Мы утверждали выше, что, если существует *какой-то* способ конкретизировать типовые переменные в терме так, чтобы он стал типизируемым, то существует и *наиболее общий* (most general), или *главный* (principal), способ это сделать. Теперь мы формализуем это утверждение.

Определение 22.5.1 Главным решением (*principal solution*) (Γ, t, S, C) называется такое решение (σ, T) , что, если (σ', T') также является решением (Γ, t, S, C) , то выполняется $\sigma \sqsubseteq \sigma'$. Если (σ, T) — главное решение, то T называется главным типом (*principal type*) терма t в контексте Γ .³

Упражнение 22.5.2 [★ →]: Найдите главный тип $\lambda x:X. \lambda y:Y. \lambda z:Z. (x z)$ ($y z$).

Теорема 22.5.3 [ГЛАВНЫЕ ТИПЫ]: Если задача (Γ, t, S, C) имеет решение, то она имеет и главное решение. Алгоритм унификации по рис. 22.2 позволяет определить, имеет ли (Γ, t, S, C) решение, и, если это так, найти главное решение.

Доказательство: По определению решения для (Γ, t, S, C) и свойствам унификации.

Следствие 22.5.4 Задача определения наличия решения y (Γ, t) разрешима.

Доказательство: По следствию 22.3.8 и теореме 22.5.3.

Упражнение 22.5.5 [РЕКОМЕНДУЕТСЯ, ★★★, →]: Сочетая алгоритмы порождения ограничений из упражнений 22.3.10 и 22.5.5, постройте программу проверки типов, вычисляющую главные типы. За основу можно взять программу *reconbase*. Типичный протокол взаимодействия с вашей программой может выглядеть так:

```

λx:X. x;

▷ <fun> : X → X

λz:ZZ. λy:YY. z (y true);

▷ <fun> : (?X₀ → ?X₁) → (Bool → ?X₀) → ?X₁

λw:W. if true then false else w false;

▷ <fun> : (Bool → Bool) → Bool

```

Типовые переменные с именами вроде $?X_0$ порождаются автоматически.

³ Не следует путать главные типы с *главной типизацией*. См. с. 367.

Упражнение 22.5.6 [★★★]: Какие сложности возникают при попытке распространения данных выше определений (22.3.2 и др.) на записи? Как можно с этими сложностями справиться?

С помощью идеи главных типов можно построить алгоритм реконструкции типов, работающий более мелкими шагами, чем описанный здесь. Вместо того, чтобы сначала порождать все ограничения, а потом пытаться их выполнить, можно чередовать порождение и разрешение ограничений так, чтобы алгоритм реконструкции типов на каждом шагу возвращал главный тип. То, что этот тип является главным, гарантирует нам, что подтерм никогда не придется анализировать повторно: алгоритм всегда берет на себя наименьшие возможные обязательства, чтобы достичь типизируемости. Существенным преимуществом такого алгоритма будет, что он сможет намного точнее выявлять ошибки в пользовательских программах.

Упражнение 22.5.7 [★★★ →]: Модифицируйте решение упражнения 22.5.5 так, чтобы унификация проводилась пошагово и возвращались главные типы.

22.6. Неявные аннотации типов

Как правило, языки с поддержкой реконструкции типов позволяют программистам полностью опускать аннотации типов на лямбда-абстракциях. Один из способов достичь этого (как мы заметили в §22.2) — заставить процедуру синтаксического анализа вставлять вместо опущенных аннотаций свежепорожденные типовые переменные. Более привлекательным решением будет добавление неаннотированных абстракций в синтаксис термов, а также введение нового правила в отношение типизации с ограничениями:

$$\frac{x \notin \mathcal{X} \quad \Gamma, x:x \vdash t_1 : T |_{\mathcal{X}} C}{\Gamma \vdash \lambda x. t_1 : x \rightarrow T |_{\mathcal{X} \cup \{x\}} C} \quad (\text{CT-AbsINF})$$

Такой подход к неаннотированным абстракциям несколько проще, чем предложение считать их синтаксическим сахаром. Кроме того, он добавляет некоторую полезную выразительность: если мы скопируем неаннотированную абстракцию несколько раз, правило CT-AbsINF позволит нам выбрать *разные* переменные в качестве типов аргумента в каждой копии. Напротив, если считать, что «голая» абстракция снабжена невидимой типовой переменной, то копирование породит несколько выражений с *одним и тем же* типом аргумента. Это различие окажется существенным при обсуждении полиморфизма через `let` в следующем разделе.

22.7. Полиморфизм через `let`

Термин *полиморфизм* (polymorphism) обозначает семейство различных механизмов, позволяющих использовать один и тот же фрагмент программы

с различными типами в различных контекстах (в §23.2 подробнее обсуждаются некоторые разновидности полиморфизма). Алгоритм реконструкции типов, приведенный выше, несложно обобщить, получая при этом простую форму полиморфизма, которая называется *полиморфизм через let* (let-polymorphism), а также полиморфизм в *стиле ML* (ML-style polymorphism), а также полиморфизм по *Дамасу-Милнеру* (Damas-Milner polymorphism). Такой полиморфизм впервые появился в исходном диалекте ML (Milner, 1978) и позднее был включен во множество удачных языков, где он служит основой мощных *обобщенных библиотек* (generic libraries) часто используемых структур (списков, массивов, деревьев, хэш-таблиц, потоков, виджетов для взаимодействия с пользователем и т. д.).

Полиморфизм через `let` можно мотивировать таким примером: допустим, что мы определяем и используем простую функцию `double`, дважды применяющую свой первый аргумент ко второму:

```
let double = λf:Nat → Nat. λa:Nat. f(f(a)) in
double (λx:Nat. succ (succ x)) 2;
```

Поскольку мы хотим применять `double` к функциям типа `Nat → Nat`, мы аннотируем её типом `(Nat → Nat) → (Nat → Nat)`. Кроме того, мы можем определить `double` так, чтобы она удваивала булевскую функцию:

```
let double = λf:Bool → Bool. λa:Bool. f(f(a)) in
double (λx:Bool. x) false;
```

Однако мы *не можем* использовать одну и ту же функцию `double` и с булевскими значениями, и с числами: если в одной и той же программе нам требуется и то, и другое, надо определить две функции, отличающиеся только аннотациями типов:

```
let doubleNat = λf:Nat → Nat. λa:Nat. f(f(a)) in
let doubleBool = λf:Bool → Bool. λa:Bool. f(f(a)) in
let a = doubleNat (λx:Nat. succ (succ x)) 1 in
let b = doubleBool (λx:Bool. x) false in ...
```

Даже если мы аннотируем абстракцию в `double` *типовoy переменной*,

```
let double = λf:X → X. λa:X. f(f(a)) in ...
```

это не поможет. Например, если мы напишем

```
let double = λf:X → X. λa:X. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ...
```

то использование `double` в определении `a` породит ограничение $X \rightarrow X = \text{Nat} \rightarrow \text{Nat}$, а использование `double` в определении `b` породит ограничение $X \rightarrow X = \text{Bool} \rightarrow \text{Bool}$. Эти ограничения выдвигают несовместимые требования к `X`, и вся программа оказывается нетипизируемой.

Где же здесь ошибка? Переменная `X` выступает в этом примере сразу в двух разных ролях. Во-первых, она отражает ограничение, по которому первый аргумент `double` при вычислении `a` должен быть функцией, типы аргумента и результата которой совпадают с типом (`Nat`) второго аргумента `double`. Во-вторых, она отражает ограничение, согласно которому аргументы `double` при

вычислении **b** также должны находиться в подобных отношениях. К сожалению, поскольку в обоих случаях используется одна и та же переменная **X**, то возникает ненужное нам ограничение, в соответствии с которым вторые аргументы при обоих вызовах **double** должны быть одного и того же типа.

Нам хотелось бы разбить эту последнюю связь — т. е. связать с разными вызовами функции **double** разные переменные **X**. К счастью, это нетрудно организовать. На первом шаге изменим обыкновенное правило типизации для **let** так, чтобы вместо вычисления типа связываемого значения **t₁** с последующим использованием его в качестве типа переменной **x** при вычислении типа тела **t₂**:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2} \quad (\text{T-LET})$$

оно *подставляло* **t₁** вместо **x** в теле, а затем проверяло типы в получившемся выражении:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2} \quad (\text{T-LETPOLY})$$

Аналогично записывается и правило типизации с ограничениями для **let**:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \mid_x C}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2 \mid_x C} \quad (\text{CT-LETPOLY})$$

В сущности, мы изменили правила типизации для **let** так, чтобы они перед определением типов производили шаг вычисления

$$\text{let } x=v_1 \text{ in } t_2 \rightarrow [x \mapsto v_1]t_2 \quad (\text{E-LETV})$$

На втором шаге перепишем определение **double**, используя *неявно аннотированные* (implicitly annotated) лямбда-абстракции из §22.6.

```
let double = λf. λa. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ...
```

Сочетание правил типизации ограничениями для **let** (CT-LETPOLY) и неявно аннотированной лямбда-абстракции (CT-ABSTRACT) дает нам именно то, что требуется: CT-LETPOLY порождает две копии определения **double**, а CT-ABSTRACT приписывает каждой из абстракций отдельную типовую переменную. Обычный процесс разрешения ограничений завершает требуемую работу.

Однако у этой схемы есть несколько недостатков, которые нужно исправить, прежде чем ее можно будет применить ее на практике. Одно очевидное упущение состоит в том, что если мы ни разу не используем связанную через **let** переменную в теле **let**, определение просто не будет подвергнуто проверке типов. Например, программа вроде

```
let x = <бессмыслица> in 5
```

пройдет проверку. Это можно исправить, добавив в правило типизации предпосылку

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \quad \Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2} \quad (\text{T-LET POLY})$$

а в правило CT-LET POLY — соответствующую предпосылку, проверяющую, что t_1 правильно типизирован.

Близкая по природе проблема состоит в том, что если тело let содержит *несколько* вхождений связанной через let переменной, то вся правая часть определения let будет проверяться при каждом вхождении, независимо от того, есть ли там неявно аннотированные лямбда-абстракции. Поскольку сама правая сторона может содержать выражения let, это правило типизации может заставить программу проверки проделать объем работы, экспоненциально зависящий от размера исходного терма!

Чтобы избежать этих постоянных перепроверок, практические реализации языков с полиморфизмом используют несколько более хитрую (но формально эквивалентную) формулировку правил типизации. Вкратце, проверка типов для терма $\text{let } x=t_1 \text{ in } t_2$ в контексте Γ происходит так:

1. С помощью правил типизации с ограничениями вычисляем тип S_1 и множество связанных с ним ограничений C_1 для связываемого терма t_1 .
2. Производим унификацию и находим наиболее общее решение σ для ограничений C_1 . Применяем σ к S_1 и Γ и получаем *главный тип* $t_1 - T_1$.
3. *Обобщаем* (generalize) остающиеся в T_1 переменные. Если остаются переменные $X_1 \dots X_n$, мы записываем главную *схему типа* (type scheme) для t_1 в виде $\forall X_1 \dots X_n . T_1$.

Здесь тонкость состоит в том, что *нельзя* обобщать переменные из T_1 , которые также встречаются в Γ , поскольку этим переменным соответствуют реальные ограничения, существующие между t_1 и его окружением. Например, в

$\lambda f:X \rightarrow X. \lambda x:X. \text{let } g=f \text{ in } g(x);$

не следует обобщать переменную X в типе $X \rightarrow X$, который приписывается терму g , поскольку это позволило бы нам писать неверные программы вроде

```
(λf:X → X. λx:X. let g=f in g(0))
  (λx:Boolean. if x then true else false)
    true;
```

4. Расширяем контекст, указывая схему типа $\forall X_1 \dots X_n . T_1$ для переменной x , и начинаем проверять типы в теле t_2 . В общем случае, теперь контекст сопоставляет каждой свободной переменной не тип, а схему типа.
5. Каждый раз, когда внутри t_2 встречается переменная x , мы рассматриваем схему ее типа $\forall X_1 \dots X_n . T_1$. Порождаем новые типовые переменные

$Y_1 \dots Y_n$ и с их помощью *конкретизируем* (instantiate) схему типа, получая $[X_1 \mapsto Y_1, \dots, X_n \mapsto Y_n]T_1$. Это последнее выражение мы используем в качестве типа x .⁴

Этот алгоритм намного эффективнее, чем простой подход с подстановкой в выражения *let* перед проверкой типов. Десятилетия использования показали, что на практике он «по существу линеен» относительно размера входной программы. Поэтому было большой неожиданностью, когда Кфури, Тюрин и Ужичин (Kfoury, Tiuryn, and Urzyczyn, 1990) и, независимо от них, Мейрсон (Mairson, 1990) показали, что в наихудшем случае сложность алгоритма по-прежнему экспоненциальна! Построенный ими пример содержит нагромождение *let* в связываемой позиции других *let* — а не в телях, где обычно встречается вложенные *let*. В результате получаются выражения, типы которых растут экспоненциально быстрее, чем сами выражения. Например, следующая программа на OCaml, приведенная Мейрсоном (Mairson, 1990), правильно типизирована, но проверка типов в ней занимает весьма долгое время:

```
let f0 = fun x → (x, x) in
  let f1 = fun y → f0(f0 y) in
    let f2 = fun y → f1(f1 y) in
      let f3 = fun y → f2(f2 y) in
        let f4 = fun y → f3(f3 y) in
          let f5 = fun y → f4(f4 y) in
            f5 (fun z → z)
```

Чтобы понять, почему это так, попробуйте ввести f_0 , f_1 и т. д. по одному в командную строку OCaml. Дальнейшее обсуждение можно найти у Кфури, Тюрина и Ужичина (Kfoury, Tiuryn, and Urzyczyn, 1994).

Последнее обстоятельство, которое следует упомянуть, заключается в том, что при проектировании языков с полиморфизмом через *let* следует внимательно обрабатывать взаимодействие полиморфизма и конструкций с побочными эффектами вроде изменяемых ячеек памяти. Эту опасность можно проиллюстрировать на простом примере:

```
let r = ref (λx. x) in
(r := (λx:Nat. succ x));
(!r) true;
```

По вышеприведенному алгоритму, правая часть *let* получает главный тип *Ref* ($X \rightarrow X$); поскольку X нигде больше не встречается, этот тип обобщается до $\forall X.\text{Ref}$ ($X \rightarrow X$), и эту схему мы присваиваем переменной r , добавляя ее в контекст. При проверке типов для присваивания во второй строке эта схема конкретизируется до *Ref* ($Nat \rightarrow Nat$). При проверке третьей строки она же конкретизируется до *Ref* ($Bool \rightarrow Bool$). Однако такая типизация некорректна, поскольку при исполнении терма произойдет попытка применения операции *succ* к значению *true*.

⁴Различие между лямбда-абстракцией, которая явно аннотирована типовой переменной, и неаннотированной абстракцией, которой алгоритм порождения ограничений присыпывает новую переменную, теряет свое значение при вводе обобщения и конкретизации. В обоих случаях связываемое выражение в *let* получает тип, включающий в себя переменную. Эта переменная подвергается обобщению, прежде чем алгоритм вносит ее в контекст, и заменяется новой переменной при каждой конкретизации.

Проблема состоит в том, что правила типизации оказались рассинхронизированы с правилами вычисления. Правила типизации, введенные в этом разделе, говорят, что как только мы видим выражение `let`, мы должны *неделенно* вставить его правую часть в тело. Однако правила вычисления говорят, что эту подстановку можно производить только *после* того, как правая часть будет сведена к значению. Правила типизации видят два употребления конструктора `ref` и анализируют их в разных окружениях, однако при вычислении выделяется только одна `ref`-ячейка.

Устранить это расхождение можно двумя способами — изменив вычисление или типизацию. В первом случае⁵ правило вычисления для `let` превратится в

$$\text{let } x=t_1 \text{ in } t_2 \rightarrow [x \mapsto t_1]t_2 \quad (\text{E-LET})$$

При такой стратегии первый шаг при вычислении нашего опасного примера заменит `x` его определением, и получится

```
(ref (λx.x)) := (λx:Nat. succ x) in
(!(ref (λx. x))) true;
```

что совершенно безопасно! В первой строке создается ячейка, инициализируемая функцией тождества, а затем в нее записывается $(\lambda x:\text{Nat}. \text{ succ } x)$. Во второй строке создается *другая* ячейка, содержащая функцию тождества, ее содержимое извлекается и применяется к аргументу `true`. Однако этот пример показывает также, что изменение правил вычисления с целью соответствовать правилам типизации дает нам язык с достаточно странной семантикой, которая уже не соответствует нашим интуитивным понятиям о порядке вычислений с вызовом по значению. (Императивные языки с другими стратегиями вычисления существуют (Augustsson, 1984), но они не смогли завоевать популярности, поскольку в них трудно понять порядок побочных эффектов во время исполнения и управлять им.)

Лучше будет изменить правило типизации так, чтобы оно соответствовало правилу вычисления. К счастью, это не представляет труда: нужно просто добавить ограничение (его часто называют *ограничение на значения*, value restriction), которое говорит, что связывание через `let` может рассматриваться как полиморфное, т. е. его переменные могут подвергаться обобщению только в случае, если его правая сторона является синтаксическим значением. Это означает, что в нашем опасном примере тип, присваиваемый переменной `x`, при добавлении в контекст равен $X \rightarrow X$, а не $\forall X. X \rightarrow X$. Ограничения, накладываемые второй строкой, заставляют `X` быть равным `Nat`, и из-за этого проверка типов в третьей строке потерпит неудачу, поскольку `Nat` и `Bool` не унифицируются.

⁵Строго говоря, в этом правиле нужно упомянуть *состояние памяти* (store), как это было сделано в главе 13, поскольку речь идет о языке со ссылками:

$$\text{let } x=t_1 \text{ in } t_2 | \mu \rightarrow [x \mapsto t_1]t_2 | \mu \quad (\text{E-LET})$$

Ограничение на значения решает нашу проблему с типовой безопасностью ценой некоторой потери выразительности: мы уже не сможем писать программы, в которых правая сторона выражения `let` выполняет некоторое интересное вычисление, и при этом ей присваивается полиморфная схема типа. Удивительно то, что это ограничение почти не влияет на практическое программирование. Райт (1995) показал это, проанализировав огромный массив кода, написанного на диалекте Standard ML в версии 1990 года (Milner, Tofte, and Harper, 1990), который поддерживал более гибкое правило типизации для `let`, основанное на *слабых типовых переменных* (weak type variables). Райт заметил, что кроме горстки случаев, правые части `let` все равно являлись значениями. Это наблюдение более или менее закрыло дискуссию, и с тех пор все основные языки с полиморфизмом в стиле ML соблюдают ограничение на значения.

Упражнение 22.7.1 [★★★ →]: Реализуйте алгоритм, наметки которого приведены в этом разделе.

22.8. Дополнительные замечания

Понятие главного типа в лямбда-исчислении восходит, по меньшей мере, к работам Карри 50-х годов (Curry and Feys, 1958). Алгоритм вычисления главных типов на основе идей Карри был построен Хиндли (Hindley, 1969); сходные алгоритмы были независимо обнаружены Моррисом (Morris, 1968) и Милнером (Milner, 1978). В области пропозициональной логики эти идеи существовали еще раньше: возможно, у Тарского в 1920-х годах, и определенно у братьев Мередит в 1950-е (Lemmon, Meredith, Meredith, Prior, and Thomas, 1957); первая их компьютерная реализация была осуществлена Дэвидом Мередитом в 1957 году. Дополнительные исторические сведения о главных типах можно найти у Хиндли (1997).

Унификация (Robinson, 1971) играет ключевую роль во многих областях информатики. Подробные введения можно найти, например, у Баадера и Нипкова (Baader and Nipkow, 1998), Баадера и Зикмана (Baader and Siekmann, 1994) или у Лассе и Плоткина (Lassez and Plotkin, 1991).

Полиморфизм через `let` в стиле ML был впервые описан Милнером (Milner, 1978). Было предложено несколько алгоритмов реконструкции типов, из которых наиболее известен Алгоритм W (Algorithm W) Дамаса и Милнера (Damas and Milner, 1982; см. также Lee and Yi, 1998). Основное отличие Алгоритма W от приведенного нами в этой главе состоит в том, что Алгоритм W приспособлен для «чистой реконструкции типов» — присвоения типов совершенно *нетипизированным* термам, — в то время как у нас происходит одновременно реконструкция и проверка типов, а термы могут содержать явные аннотации, где могут (но не обязательно должны) встречаться переменные. Это усложняет наше описание с технической точки зрения (особенно доказательство полноты в теореме 22.3.7, где нам приходится заботиться, чтобы переменные, введенные программистом, отличались от тех, которые появляются в результате правил порождения ограничений), однако такое изложение лучше сочетается со стилем повествования из других глав книги.

Классическая статья Карделли (Cardelli, 1987) перечисляет несколько вопросов, возникающих при реализации алгоритмов. Кроме того, описание алгоритмов реконструкции типов можно найти у Аппеля (Appel, 1998), Ахо и др. (Aho, Sethi, and Ullman, 1986), а также у Рида (Reade, 1989). Чрезвычайно изящное изложение базовой системы, называемой *мини-ML* (mini-ML) (Clement, Despeugoux, Despeugoux, and Kahn, 1986) часто служит основой для теоретических рассуждений. Тюрин (Tiuryn, 1990) обозревает широкий спектр проблем реконструкции типов.

Главные типы не следует путать с родственным понятием *главной типизации* (principal typing). Разница состоит в том, что при вычислении главного типа контекст Γ и терм t рассматриваются как входы алгоритма, а главный тип T служит его выходом. Алгоритм для вычисления главной типизации принимает на входе только t , а на выходе выдает как Γ , так и T — т. е. он вычисляет *минимальные предположения* (minimal assumptions) о свободных переменных в t . Главные типизации оказываются полезны при раздельной компиляции и «оптимальной перекомпиляции», для пошагового вывода типов и поиска ошибок типизации. К сожалению, во многих языках, включая ML, имеются главные типы, но не главные типизации. См. Jim, 1996.

Полиморфизм в стиле ML, сочетающий мощность и простоту, оказывается «наилучшим компромиссом» в пространстве возможных языковых проектов; его смешение с другими мощными конструкциями типизации часто бывает весьма нетривиальным. Наиболее успешный результат в этой области получен в изящном описании реконструкции типов для записей, предложенном Вандом (Wand, 1987) и развитым в работах Ванда (Wand, 1988, 1989b), Реми (Rémy, 1989, 1990; Rémy, 1992a,b, 1998) и многих других. Идея состоит в том, чтобы ввести новую разновидность переменных — *строчные переменные* (row variables), которые имеют значением не типы, а целые «строки», состоящие из меток и связанных с ними типов. Для решения множеств ограничений, связанных с переменными строк, используется простая форма *эквациональной унификации* (equational unification); см. упражнение 22.5.6. Гарриг (Garrigue and Aït-Kaci, 1994) и другие разработали аналогичные методы для вариантных типов. Эти методы были распространены на общие понятия *классов типов* (type classes) (Kaes, 1988; Wadler and Blott, 1989), *типов ограничений* (constraint types) (Odersky, Sulzmann, and Wehr, 1999) и *специфицированных типов* (qualified types) (Jones, 1994b,a), которые лежат в основе системы *классов типов* в языке Haskell (Hall et al., 1996; Hudak et al., 1992; Thompson, 1999); подобные идеи встречаются также в языках Mercury (Somogyi, Henderson, and Conway, 1996) и Clean (Plasmeijer, 1998).

Уэллс (Wells, 1994) показал, что задача реконструкции типов для более мощной формы *импредикативного полиморфизма* (impredicative polymorphism) неразрешима. Некоторые разновидности *частичной реконструкции типов* (partial type reconstruction) для этой системы также оказываются неразрешимыми. В §23.6 и §23.8 приводится дополнительная информация об этих результатах, и о том, как реконструкция типов в стиле ML сочетается с более сильными формами вроде *полиморфизма ранга 2* (rank-2 polymorphism).

Относительно сочетания реконструкции типов в стиле ML с подтипа-

ми были опубликованы некоторые многообещающие результаты (Aiken and Wimmers, 1993; Eifrig, Smith, and Trifonov, 1995; Jagannathan and Wright, 1995; Trifonov and Smith, 1996; Odersky, Sulzmann, and Wehr, 1999; Flanagan and Felleisen, 1997; Pottier, 1997), однако практические реализации пока не нашли широкого применения.

Было показано, что распространение реконструкции типов по ML на рекурсивные типы (глава 20) *не представляет* серьезных трудностей (Huet, 1975, 1976). Единственное существенное отличие от алгоритмов, приведенных в этой главе, касается определения унификации, в котором опускают проверку на вхождение (occur check) (которая обычно проверяет, что подстановка, которую возвращает алгоритм унификации, не содержит циклов). После этого, чтобы по-прежнему гарантировать завершение алгоритма, требуется изменить представление данных и обеспечить совместное использование подструктур, например, через деструктивные обновления (возможно, циклических) структур, состоящих из указателей. Такие представления обычны в высокопроизводительных реализациях языков.

С другой стороны, сочетание реконструкции типов с рекурсивно определенными *термами* порождает сложную проблему, известную под названием *полиморфной рекурсии* (polymorphic recursion). Простое (и не вызывающее проблем) правило типизации для рекурсивных определений функций в ML говорит, что рекурсивная функция может употребляться внутри своего определения только мономорфно (т. е. все рекурсивные вызовы должны иметь одни и те же типы аргументов и результатов), тогда как в остальной программе ее можно использовать полиморфно (с аргументами и результатами различных типов). Майкрофт (Mycroft, 1984) и Мейртенс (Meertens, 1983) предложили полиморфное правило типизации, позволяющее конкретизировать различными типами рекурсивные вызовы рекурсивной функции из ее собственного тела. Было показано, что в этом расширении, известном как *Исчисление Милнера-Майкрофта* (Milner-Mycroft calculus), задача реконструкции типов неразрешима (Henglein, 1993, и, независимо от него, Kfoury, Tiuryn, and Urzyczyn, 1993a); оба эти доказательства зависят от неразрешимости (неограниченной) задачи полу-унификации, как показано Кфури, Тюриным и Ужичиным (Kfoury, Tiuryn, and Urzyczyn, 1993b).

Глава 23

Универсальные типы

В предыдущей главе мы рассмотрели простую разновидность *полиморфизма через let*, имеющегося в ML. В этой главе мы изучаем более общую форму полиморфизма в рамках мощного исчисления, известного как *Система F* (System F).

23.1. Мотивация

Как мы упомянули в §22.7, в простом типизированном лямбда-исчислении можно написать бесконечное число «удваивающих» функций:

```
doubleNat = λf:Nat → Nat. λx:Nat. f (f x);
doubleRcd = λf:{l:Bool} → {l:Bool}. λx:{l:Bool}. f (f x);
doubleFun = λf:(Nat → Nat) → (Nat → Nat). λx:Nat → Nat. f (f x);
```

Каждая из этих функций применима к своему типу аргумента, но их поведение одинаково (более того, текст функций, помимо аннотаций типа, тоже одинаков). Если нам нужно применять операции удвоения к различным типам аргументов внутри одной программы, нам придется написать отдельные определения `doubleT` для каждого `T`. Такое программирование путем копирования и вставки текста нарушает один из главных принципов разработки программ:

ПРИНЦИП АБСТРАКЦИИ: Каждая существенная область функциональности в программе должна быть реализована всего в одном месте программного кода. Если различные фрагменты кода реализуют аналогичную функциональность, как правило, имеет смысл слить их в один фрагмент, *абстрагируя* (*abstracting out*) различающиеся части.

На протяжении большей части этой главы изучается чистая Система F (рис. 23.1); в примерах из §23.4 используются расширения этой системы различными уже известными нам конструкциями. Соответствующая реализация на OCaml называется `fullpoly`. (Примеры с использованием пар и списков требуют интерпретатора `fullomega`.)

В данном случае различающиеся части — это типы! Таким образом, нам нужен способ абстрагировать тип терма, а затем конкретизировать абстрактный терм аннотациями нужного типа.

23.2. Разновидности полиморфизма

Системы типов, позволяющие использовать единый фрагмент кода с различными типами, известны под общим названием *полиморфных* (polymorphic) (*поли* = много, *морф* = форма). В современных языках встречается несколько разновидностей полиморфизма (приводимая здесь классификация основана на работах Стрейчи (Strachey, 1967), а также Карделли и Вегнера (Cardelli and Wegner, 1985)).

Параметрический полиморфизм (parametric polymorphism), которому посвящена эта глава, позволяет давать участку кода «обобщенный» тип, используя переменные вместо настоящих типов, а затем конкретизировать, замещая переменные типами. Параметрические определения *однородны*: все экземпляры данного фрагмента кода ведут себя одинаково.

Наиболее мощной формой параметрического полиморфизма является *импредикативный* (impredicative) полиморфизм, или *полиморфизм первого класса* (first-class polymorphism). Именно он обсуждается в данной главе. На практике больше распространен полиморфизм в *стиле ML* (ML-style), то есть *полиморфизм через let* (let-polymorphism), ограниченный связываниями *let* верхнего уровня и запрещающий функции, в которых аргументами служат полиморфные значения. Взамен он предлагает удобную и естественную форму *реконструкции типов* (type reconstruction) (глава 22). Параметрический полиморфизм первого класса приобретает популярность в языках программирования. Он служит техническим основанием для мощной системы модулей в языках вроде ML (см. Harper and Stone, 2000).

Специализированный полиморфизм (ad-hoc polymorphism), наоборот, позволяет полиморфному значению демонстрировать различное поведение при его «рассмотрении» относительно разных типов. Самый обычный пример специализированного полиморфизма — *перегрузка* (overloading), когда один и тот же символ функции соответствует различным реализациям; компилятор (или система времени выполнения, в зависимости от того, идет ли речь о *статическом* (static) или *динамическом* (dynamic) разрешении перегрузки) выбирает подходящую реализацию для каждого случая применения функции, исходя из типов ее аргументов.

Обобщая перегрузку функций, мы получаем диспетчеризацию *мультиметодов* (multi-method), как в CLOS (Bobrow et al., 1988; Kiczales et al., 1991) и Cecil (Chambers, 1992; Chambers and Leavens, 1994). Этот механизм был formalized в λ -&-исчислении Кастаньи, Гелли и Лонго (Castagna, Ghelli, and Longo, 1995; см. Castagna,

1997).

Более мощная форма специализированного полиморфизма — *интенсиональный полиморфизм* (intensional polymorphism) (Harper and Morrisett, 1995; Crary, Weirich, and Morrisett, 1998), который допускает ограниченные вычисления над типами во время выполнения. Интенсиональный полиморфизм лежит в основе некоторых сложных методов оптимизации для полиморфных языков, в том числе бестеговой сборки мусора, «распакованных» аргументов функций, полиморфной сериализации, а также оптимизированных по памяти «плоских» структур данных.

Еще более мощные формы специализированного полиморфизма можно получить на основе конструкции `typecase`, которая позволяет осуществлять произвольное сопоставление информации о типах с образцом во время выполнения (Abadi, Cardelli, Pierce, and Rémy, 1995; Abadi, Cardelli, Pierce, and Plotkin, 1991b; Henglein, 1994; Leroy and Mauny, 1991; Thatte, 1990). Такие конструкции, как `instanceof` в Java, можно рассматривать как ограниченную разновидность `typecase`.

Полиморфизм через подтипы (subtype polymorphism) (глава 15) присваивает несколько типов одному терму с помощью правила включения и позволяет избирательно «забывать» часть информации о поведении терма.

Эти категории не являются взаимоисключающими: различные формы полиморфизма могут смешиваться в одном языке. Например, Standard ML поддерживает как параметрический полиморфизм, так и простую перегрузку встроенных арифметических операций, но не поддерживает подтипы, а Java содержит подтипы, перегрузку и простой специализированный полиморфизм (`instanceof`), но не содержит (на момент написания книги) параметрический полиморфизм. Существует несколько предложений по добавлению параметрического полиморфизма в Java; наиболее известен из них GJ (Bracha, Odersky, Stoutamire, and Wadler, 1998).

Использование термина «полиморфизм» без уточнений приводит к некоторой путанице при общении различных программистских сообществ. Среди функциональных программистов (т. е. тех, кто использует или проектирует языки вроде ML, Haskell и т. п.), он почти всегда означает параметрический полиморфизм. Среди программистов, работающих с объектно-ориентированными языками, он, напротив, почти всегда означает полиморфизм через подтипы, а параметрический полиморфизм обозначается термином *обобщенные функции* (generics).

23.3. Система F

Система, которую мы изучаем в этой главе, известна под названием *Системы F* (System F). Ее впервые открыл Жан-Ив Жирар (Girard, 1972) в контексте теории доказательств в логике. Несколько позже систему типов с практически той же выразительной мощностью независимо построил специа-

лист по информатике Джон Рейнольдс (Reynolds, 1974), который назвал ее *полиморфным лямбда-исчислением* (polymorphic lambda-calculus). Эта система активно используется как исследовательский инструмент для работ по основаниям полиморфизма, а также послужила базой для многочисленных проектов языков программирования. Иногда ее также называют *лямбда-исчислением второго порядка* (second-order lambda calculus), поскольку по *соотношению Карри-Говарда* (Curry-Howard correspondence) она аналогична интуиционистской логике второго порядка, в которой разрешена квантификация не только по отдельным объектам [термам], но и по предикатам [типам].

Определение Системы F является естественным расширением λ_{\rightarrow} , просто-го типизированного лямбда-исчисления. В λ_{\rightarrow} , лямбда-абстракция служит для абстрагирования термов из термов, а с помощью применения вместо абстрагированных частей подставляются значения. Поскольку теперь нам понадобился механизм для абстрагирования *типов* из термов, а также для последующего заполнения абстракций, мы вводим новую форму абстракции, которую мы записываем в виде $\lambda X.t$, параметром которой служит тип, и новую форму применения, $t[T]$, в которой аргументом служит выражение типа. Мы называем эти новые абстракции *абстракциями типа* (type abstractions), а новую конструкцию применения — *применением* (type application) или *конкретизацией типа* (type instantiation).

Когда при вычислении абстракция типа сочетается с применением типа, получается редекс, в точности как в λ_{\rightarrow} . Мы добавляем в систему правило редукции:

$$(\lambda X.t_{12})[T_2] \rightarrow [X \mapsto T_2]t_{12} \quad (\text{E-TAPP TABS})$$

аналогичное обыкновенному правилу редукции для термов-абстракций и термов-применений:

$$(\lambda x:T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APP ABS})$$

Например, при применении полиморфной *функции тождества* (identity function)

`id = $\lambda X. \lambda x:X. x$;`

к типу Nat в выражении `id [Nat]` результатом будет $[X \mapsto \text{Nat}](\lambda x:X.x)$, т. е. $\lambda x:\text{Nat}.x$, функция тождества на натуральных числах.

Наконец, мы должны указать *тип* полиморфной абстракции. Такие типы, как $\text{Nat} \rightarrow \text{Nat}$, используются для описания обычных функций вроде $\lambda x:\text{Nat}.x$; теперь для описания полиморфных функций вроде `id` нам нужен другой вид «функционального типа», областью определения которого служат типы. Заметим, что для каждого типа-аргумента T , к которому применяется `id`, она возвращает функцию вида $T \rightarrow T$; то есть, тип результата `id` зависит от конкретного типа, передаваемого ей в качестве аргумента. Чтобы отразить эту зависимость, мы записываем тип `id` как $\forall X.X \rightarrow X$. Правила типизации для полиморфных абстракций и применения аналогичны правилам для абстракций и применения на уровне термов.

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \quad (\text{T-TABS})$$

$$\frac{\Gamma \vdash t_1 : \forall X. T_{12}}{\Gamma \vdash t_1 [T_2] : [X \mapsto T_2] T_{12}} \quad (\text{T-TAPP})$$

Обратите внимание, что мы включаем типовую переменную X в контекст, используемый в подвыводе для t . Продолжает действовать соглашение (5.3.4), согласно которому имена переменных (термовых или типовых) должны быть выбраны так, чтобы отличаться от уже связанных в Γ , а переменные, связанные лямбдами, можно при необходимости переименовывать, чтобы выполнить это условие. (В некоторых вариантах изложения Системы F это условие новизны задается как явное дополнительное условие в правиле T-TABs, а не встраивается в правила построения контекстов, как у нас.) Пока что единственная роль типовых переменных в контекстах заключается в отслеживании областей видимости и защите от повторного добавления одной и той же типовой переменной в контекст. В последующих главах мы будем снабжать типовые переменные различной информацией, например, *ограничениями* (bounds) (глава 26) или *видами* (kinds) (глава 29).

На рис. 23.1 приведено полное определение полиморфного лямбда-исчисления, и выделены его отличия от $\lambda\rightarrow$. Как обычно, здесь определяется только чистое исчисление, без конструкторов типов вроде записей, без базовых типов вроде `Nat` и `Bool` и без расширений термового языка вроде `let` и `fix`. Эти дополнительные конструкции можно без труда добавить в чистую систему, и в последующих примерах мы будем их часто использовать.

23.4. Примеры

Приведем несколько примеров программирования с использованием полиморфизма. Начнем, для разминки, с нескольких небольших, но постепенно усложняющихся примеров, которые иллюстрируют выразительные возможности Системы F. Затем кратко рассмотрим основные идеи «обыкновенного» полиморфного программирования со списками, деревьями и т. п. В последних двух разделах вводятся типизированные варианты кодирования по Чёрчу для простых алгебраических типов — булевых значений, чисел и списков. (Само кодирование рассматривалось в главе 5 для бестипового лямбда-исчисления.) Несмотря на то, что эти варианты кодирования не имеют особой практической ценности — для этих важных элементов языка проще создать хороший код, если они встроены как примитивы, — они служат замечательными примерами, на которых можно продемонстрировать как тонкости, так и выразительную силу Системы F. В главе 24 мы увидим некоторые другие приложения полиморфизма в области *модульного программирования* (modular programming) и *абстрактных типов данных* (ATD).

Разминка

Мы уже показали, как с помощью абстракции и применения типов можно определить единую полиморфную функцию тождества

```
id = λx. λx:X. x;
```

```
> id : ∀X. X → X
```

и конкретизировать ее так, чтобы получить любую необходимую функцию тождества:

```
id [Nat];  
  
> <fun> : Nat → Nat  
  
id [Nat] 0;  
  
> 0 : Nat
```

Более полезным примером может служить полиморфная функция удвоения:

```
double = λX. λf:X→X. λa:X. f (f a);  
  
> double : ∀X. (X → X) → X → X
```

Абстракция по типу X позволяет получать функции удвоения для конкретных типов, применяя функцию `double` к различным аргументам-типам:

```
doubleNat = double [Nat];  
  
> doubleNat : (Nat → Nat) → Nat → Nat  
  
doubleNatArrowNat = double [Nat → Nat];  
  
> doubleNatArrowNat : ((Nat → Nat) → Nat → Nat) →  
                           (Nat → Nat) → Nat → Nat
```

Конкретизировав функцию `double` аргументом-типом, её можно применить к конкретной функции и аргументу, имеющим соответствующие типы:

```
double [Nat] (λx:Nat. succ(succ(x))) 3;  
  
> 7 : Nat
```

Вот пример похитрее: полиморфное самоприменение. Напомним, что в простом типизированном лямбда-исчислении невозможно присвоить тип терму $\lambda x. x x$ (упражнение 9.3.2). А вот в Системе F этот терм оказывается типизируем, если придать x полиморфный тип и должным образом его конкретизировать:

```
selfApp = λx:∀X.X→X. x [∀X.X→X] x;  
  
> selfApp : (forall X. X → X) → (forall X. X → X)
```

В качестве (чуть) более полезного примера самоприменения можно применить к себе самой функцию `double` и получить полиморфную функцию учетверения:

```
quadruple = λX. double [X→X] (double [X]);  
  
> quadruple : ∀X. (X → X) → X → X
```

Упражнение 23.4.1 [★ →]: Используя правила по рис. 23.1, убедитесь в том, что вышеприведенные термы на самом деле имеют типы, которые указаны в примерах.

Полиморфные списки

Практическое программирование с использованием полиморфизма, как правило, намного прозаичнее, чем хитрые примеры из предыдущего подраздела. В качестве примера обыкновенного полиморфного программирования, предположим, что в нашем языке есть конструктор типов `List` и конструкто-ры термов для обычных примитивов работы со списками, имеющие следую-щие типы:

```
> nil : ∀X. List X
  cons : ∀X. X → List X → List X
  isnil : ∀X. List X → Bool
  head : ∀X. List X → X
  tail : ∀X. List X → List X
```

Когда мы впервые ввели списки в §11.12, мы позволили применять элементарные операции к спискам с элементами любого типа через «специализирован-ные» правила вывода, созданные специально для списков. Теперь мы можем приписать этим операциям полиморфные типы, выраждающие в точности те же самые ограничения — то есть, списки больше не надо «вплывать» в базовый язык; их можно просто рассматривать как библиотеку, содержащую несколько констант, имеющих определенные полиморфные типы. То же самое верно для типа `Ref` и элементарных операций над ссылочными ячейками из главы 13, а также для многих других структур данных и управления.

С помощью этих примитивов мы можем определять собственные полиморфные операции над списками. Вот, например, полиморфная функция `map`, которая принимает функцию, переводящую `X` в `Y`, и список `X`-ов, и возвращает список `Y`-ов.

```
map = λX. λY.
  λf: X → Y.
  (fix (λm: (List X) → (List Y).
    λl: List X.
      if isnil [X] l
      then nil [Y]
      else cons [Y] (f (head [X] l))
        (m (tail [X] l))));

> map : ∀X. ∀Y. (X → Y) → List X → List Y

l = cons [Nat] 4 (cons [Nat] 3 (cons [Nat] 2 (nil [Nat])));
> l : List Nat

head [Nat] (map [Nat] [Nat] (λx:Nat. succ x) l);
> 5 : Nat
```

Упражнение 23.4.2 [★ →]: Убедитесь в том, что `map` действительно имеет указанный тип.

Упражнение 23.4.3 [РЕКОМЕНДУЕТСЯ, ★★]: Используя в качестве образца *map*, напишите полиморфную функцию обращения списка

```
reverse : ∀X. List X → List X
```

Это упражнение лучше всего выполнять в интерактивном режиме. Запустите интерпретатор *fullomega* и скопируйте содержимое файла *test.f* из каталога *fullomega* в начало своего исходного файла. (В этом файле содержатся определения конструктора *List* и соответствующих операций. Эти определения требуют мощных возможностей абстракции Системы F_ω , описанных в главе 29. Для работы с этим упражнением не обязательно в точности понимать, как они работают.)

Упражнение 23.4.4 [★★ →]: Напишите простую полиморфную функцию сортировки

```
sort : ∀X. (X → X → Bool) → (List X) → List X
```

в которой первым аргументом служит функция сравнения для элементов типа X .

Кодирование по Чёрчу

В §5.2 мы видели, что многие элементарные значения данных, такие как булевские значения, числа и списки, могут быть закодированы в виде функций в чистом бестиповом лямбда-исчислении. В этом разделе мы покажем, как то же самое *кодирование по Чёрчу* (Church encoding) можно произвести в Системе F. Читатель может при желании освежить свое понимание кодирования по Чёрчу, перечитав §5.2.

Кодирование по Чёрчу интересно по двум причинам. Во-первых, оно служит хорошим упражнением, развивающим понимание абстракции и применения типов. Во-вторых, оно показывает, что, как и чистое бестиповое лямбда-исчисление, Система F является вычислительно весьма богатым языком в том смысле, что чистая система способна выразить широкий спектр структур данных и управления. Следовательно, если мы разработаем полноценный язык программирования, в основе которого лежит Система F, то все эти структуры можно добавить в него как примитивы (из соображений эффективности, а также для того, чтобы снабдить их более удобным конкретным синтаксисом), не нарушив при этом основные свойства базового языка. Разумеется, для всех интересных высокогородневых языковых конструкций это не так. Например, добавление *ссылок* (references) в Систему F, как мы это сделали для λ_\rightarrow в главе 13, влечет за собой существенное изменение фундаментальной вычислительной природы исчисления.

Начнем с Чёрчевых булевских значений. Напомним, что в бестиповом лямбда-исчислении мы представляли булевские константы *true* и *false* в виде лямбда-термов *tru* и *fls*:

```
tru = λt. λf. t;
fls = λt. λf. f;
```

Каждый из этих термов принимает два аргумента, и возвращает один из них. Если мы хотим присвоить `tru` и `f1s` общий тип, нам нужно предположить, что оба аргумента имеют один и тот же тип (поскольку вызывающая процедура не знает, имеет ли она дело с `tru` или с `f1s`), но этот тип может быть каким угодно (поскольку `tru` и `f1s` со своими аргументами ничего не делают, а только возвращают один из них). Эти соображения приводят к следующему типу для `tru` и `f1s`:

```
CBool = ∀X. X → X → X
```

Термы `tru` и `f1s` в Системе F получаются добавлением соответствующих аннотаций типа к вышеприведенным бестиповым версиям:

```
tru = λX. λt:X. λf:X. t;
> tru : CBool
f1s = λX. λt:X. λf:X. f;
> f1s : CBool
```

Стандартные булевские операции, такие как `not`, можно выразить, порождая новое булевское значение, которое при помощи имеющегося решает, который из аргументов требуется вернуть:

```
not = λb:CBool. λX. λt:X. λf:X. b [X] f t;
> not : CBool → CBool
```

Упражнение 23.4.5 [РЕКОМЕНДУЕТСЯ, ★]: *Напишите терм, принимающий два аргумента типа CBool и возвращающий их конъюнкцию (оператор and).*

Аналогично можно обращаться и с числами. Числа Чёрча (Church numerals), введенные в §5.2, кодируют натуральное число `n` в виде функции, которая принимает два аргумента, `s` и `z`, и `n` раз применяет `s` к `z`:

```
c0 = λs. λz. z;
c1 = λs. λz. s z;
c2 = λs. λz. s (s z);
c3 = λs. λz. s (s (s z));
```

Понятно, что аргумент `z` должен иметь тот же тип, что и область определения `s`, и что результат `s` должен снова иметь тот же самый тип. Это приводит нас к следующему типу чисел Чёрча для Системы F:

```
CNat = ∀X. (X → X) → X → X;
```

Элементы этого типа получаются путем добавления соответствующих аннотаций к бестиповым числам Чёрча:

```
c0 = λX. λs:X → X. λz:X. z;
> c0 : CNat
```

```
c1 = λX. λs:X → X. λz:X. s z;
▷ c1 : CNat
c2 = λX. λs:X → X. λz:X. s (s z);
▷ c2 : CNat
```

Типизированная функция-последователь для чисел Чёрча определяется так:

```
csucc = λn:CNat. λX. λs:X → X. λz:X. s (n [X] s z);
▷ csucc : CNat → CNat
```

То есть, `csucc` возвращает элемент типа `CNat`, который, получая `s` и `z`, `n` раз применяет `s` к `z` (путем вызова `n` с аргументами `s` и `z`), а затем применяет `s` еще раз. Подобным образом можно определить и другие арифметические операции. Например, сложение можно определить либо через `csucc`:

```
cplus = λm:CNat. λn:CNat. m [CNat] csucc n;
▷ cplus : CNat → CNat → CNat
либо более прямым образом:
cplus = λm:CNat. λn:CNat. λX. λs:X → X. λz:X. m [X] s (n [X] s z);
▷ cplus : CNat → CNat → CNat
```

Если в нашем языке присутствуют также числа в виде примитивов (рис. 8.2), то мы можем преобразовывать числа Чёрча в обычные с помощью следующей функции:

```
cnat2nat = λm:CNat. m [Nat] (λx:Nat. succ(x)) 0;
▷ cnat2nat : CNat → Nat
Она позволяет убедиться в том, что наши операции над числами Чёрча действительно вычисляют требуемые арифметические функции:
cnat2nat (cplus (csucc c0) (csucc (csucc c0)));
```

```
▷ 3 : Nat
```

Упражнение 23.4.6 [РЕКОМЕНДУЕТСЯ, ★★]: Напишите функцию `iszzero`, которая будет возвращать `true`, будучи применена к числу Чёрча `c0`, и `false` в противном случае.

Упражнение 23.4.7 [★★ →]: Убедитесь в том, что термы

```
ctimes = λm:CNat. λn:CNat. λX. λs:X → X. n [X] (m [X] s);
▷ ctimes : CNat → CNat → CNat
cexp = λm:CNat. λn:CNat. λX. n [X → X] (m [X]);
▷ cexp : CNat → CNat → CNat
```

имеют указанные типы. Неформально объясните, как они реализуют арифметическое умножение и возвведение в степень.

Упражнение 23.4.8 [РЕКОМЕНДУЕТСЯ, ★★]: Покажите, что тип

$\text{PairNat} = \forall X. (\text{CNat} \rightarrow \text{CNat} \rightarrow X) \rightarrow X;$

можно использовать для представления пар чисел, написав функции

```
pairNat : CNat → CNat → PairNat;
fstNat : PairNat → CNat;
sndNat : PairNat → CNat;
```

для построения элемента этого типа из пары чисел, а также для доступа к первой и второй компоненте такой пары.

Упражнение 23.4.9 [РЕКОМЕНДУЕТСЯ, ★★★]: С помощью функций, определенных в упр. 23.4.8, напишите функцию *pred*, вычисляющую предшественник числа Чёрча (и возвращающую 0, если 0 подан на вход). Подсказка: основная идея приведена в примере из §5.2. Определите функцию *f* : $\text{PairNat} \rightarrow \text{PairNat}$, которая переводит пару (i, j) в $(i+1, i)$ — то есть, отбрасывает второй компонент своего аргумента, копирует первый компонент во второй, и увеличивает первый. Тогда при применении *f* к начальной паре $(0, 0)$ дадут нам пару $(n, n - 1)$, откуда мы можем получить предшественника *n*, взяв второй компонент.

Упражнение 23.4.10 [★★★]: Вот еще один способ получить предшественник числа Чёрча. Пусть *k* обозначает бестиповый лямбда-терм $\lambda x. \lambda y. x$, а *i* обозначает $\lambda x. x$. Бестиповый лямбда-терм

```
vpred = λn. λs. λz. n (λp. λq. q (p s)) (k z) i
```

(из статьи Барендрегта (*Barendregt, 1992*), со ссылкой на Й. Вельманса) вычисляет предшественник данного бестипового числа Чёрча. Покажите, что этот терм можно типизировать в Системе F, добавляя по необходимости абстракции и применения типов, а также помечая связанные переменные в бестиповом терме соответствующими типами. Дополнительные очки наберет тот, кто объяснит, как этот терм действует!

Кодирование списков

В качестве последнего примера распространим кодирование чисел по Чёрчу на случай списков. Этот пример демонстрирует изящество и мощь Системы F, поскольку показывает, что все примеры программ из предыдущего подраздела, касающиеся полиморфной обработки списков, выражимы в чистом языке. (Удобства ради, мы порождаем рекурсивные функции общего вида через конструкцию *fix*, однако в сущности, те же построения могут работать и без нее. См. упражнения 23.4.11 и 23.4.12.)

В упражнении 5.2.8 мы убедились, что в бестиповом лямбда-исчислении списки можно представить при помощи приемов, очень близких тем, которые используются при кодировании натуральных чисел. В сущности, число в единичной записи подобно списку из некоторых элементов, точная природа

которых несущественна. Обобщая эту идею на элементы произвольного типа, мы получаем способ кодирования списков по Чёрчу, в котором список с элементами x , y и z представляется как функция, которая, получая функцию f и начальное значение v , вычисляет $f\ x\ (f\ y\ (f\ z\ v))$. В терминологии OCaml, список представляется в виде своей собственной функции `fold_right`.

Тип `List X` списков с элементами типа X определяется так:

```
List X = ∀R. (X → R → R) → R → R;
```

Значение `nil` в этом представлении записать несложно:¹

```
nil = λX. (λR. λc:X → R → R. λn:R. n) as List X;
```

```
▷ nil : ∀X. List X
```

Операции `cons` и `isnil` также не представляют труда:

```
cons = λX. λhd:X. λtl:List X.
        (λR. λc:X → R → R. λn:R. c hd (tl [R] c n)) as List X;
```

```
▷ cons : ∀X. X → List X → List X
```

```
isnil = λX. λl:List X. l [Bool] (λhd:X. λtl:Bool. false) true;
```

```
▷ isnil : ∀X. List X → Bool
```

Операция `head` требует несколько больших усилий. Первая сложность состоит в вопросе, что делать с головой пустого списка. На этот вопрос мы можем ответить, вспомнив, что при наличии в языке оператора неподвижной точки общего вида с его помощью можно построить выражение любого типа. На самом деле, с помощью абстракции типов мы можем пойти еще дальше и написать единую обобщенную функцию, которая, получая тип X , выдает функцию из `Unit` в X , зацикливающуюся при применении к `unit`.

```
diverge = λX. λ_:Unit. fix (λx:X. x);
```

```
▷ diverge : ∀X. Unit → X
```

Теперь мы можем использовать `diverge[X]` `unit` в качестве «результата» вычисления `head [X]` `nil`.

```
head = λX. λl:List X. l [X] (λhd:X. λtl:X. hd) (diverge [X] unit);
▷ head : ∀X. List X → X
```

К сожалению, это определение — еще не вполне то, что нам нужно: функция не завершается *никогда*, даже будучи примененной к непустым спискам. Чтобы получить требуемое поведение, нужно несколько изменить функцию так, чтобы `diverge [X]` не получала свой аргумент типа `Unit`, выступая в качестве аргумента `l`. Ради этого мы убираем аргумент `unit` и соответствующим образом изменяем тип первого аргумента `l`:

¹Аннотация `as` помогает программе проверки типов вывести тип `nil` в читаемом виде. Как мы видели в §11.4, все программы проверки типов из этой книги перед печатью типа проводят простую операцию «схлопывания» сокращений. Однако функция схлопывания недостаточно умна, чтобы справиться с такими «параметрическими сокращениями», как `List`.

```

head =
  λX. λl: List X.
    (l [Unit → X] (λhd:X. λtl:Unit → X. λ_:Unit. hd) (diverge [X]))
      unit;

▷ head : ∀X. List X → X

```

То есть, `l` применяется к функции типа $X \rightarrow (\text{Unit} \rightarrow X) \rightarrow (\text{Unit} \rightarrow X)$ и к базовому значению типа $\text{Unit} \rightarrow X$. В случае, когда `l` соответствует пустому списку, результатом будет `diverge [X]`; однако когда `l` соответствует непустому списку, результатом будет функция, которая принимает `unit` и возвращает головной элемент `l`. Результат `l` в конце применяется к `unit`, и таким образом извлекается настоящая голова списка, имеющая нужный нам тип (или, если нам не повезло, вычисление зацикливается). Таким образом, у `head` оказывается нужный нам тип.

В случае функции `tail` мы используем сокращение `Pair X Y` (обобщение типа `PairNat` из Упражнения 23.4.8) и с его помощью строим кодирование по Чёрчу пар, у которых первый элемент имеет тип `X`, а второй — тип `Y`:

```
Pair X Y = ∀R. (X → Y → R) → R;
```

Операции над парами являются простыми обобщениями вышеприведенных операций над типом `PairNat`:

```

▷ pair : ∀X. ∀Y. X → Y → Pair X Y
  fst : ∀X. ∀Y. Pair X Y → X
  snd : ∀X. ∀Y. Pair X Y → Y

```

Теперь функция `tail` пишется так:

```

tail =
  λX. λl: List X.
    (fst [List X] [List X] (
      l [Pair (List X) (List X)]
        (λhd: X. λtl: Pair (List X) (List X).
          pair [List X] [List X]
            (snd [List X] [List X] tl)
              (cons [X] hd (snd [List X] [List X] tl)))
        (pair [List X] [List X] (nil [X]) (nil [X]))));

```

```
▷ tail : ∀X. List X → List X
```

Упражнение 23.4.11 [★★]: Строго говоря, в примерах из этого подраздела используется не чистая Система F, поскольку для порождения значения, «возвращаемого», когда `head` применяется к пустому списку, мы использовали `fix`. Напишите альтернативную версию `head`, которая принимает дополнительный параметр и возвращает его (вместо зацикливания), если список на входе оказывается пустым.

Упражнение 23.4.12 [РЕКОМЕНДУЕТСЯ, ★★★]: Напишите на языке чистой Системы F (без использования `fix`) функцию `insert`, имеющую тип

```
∀X. (X → X → Bool) → List X → X → List X
```

которая принимает функцию сравнения, отсортированный список, и новый элемент. Она должна вставить элемент в список на подходящее ему место (т. е. вслед за всеми элементами, меньшими его самого). При помощи *insert* напишите функцию сортировки для списков на языке чистой Системы F.

23.5. Основные свойства

Фундаментальные свойства Системы F весьма близки к свойствам простого типизированного лямбда-исчисления. В частности, доказательства теорем о сохранении типов и о продвижении являются простыми расширениями доказательства тех же теорем из главы 9.

Теорема 23.5.1 [Сохранение]: *Если $\Gamma \vdash t : T$ и $t \rightarrow t'$, то $\Gamma \vdash t' : T$.*
Доказательство: Упражнение [Рекомендуется, ★★★].

Теорема 23.5.2 [Продвижение]: *Если имеется замкнутый правильно типизированный терм t , то t либо является значением, либо существует некоторый терм t' такой, что $t \rightarrow t'$.*

Доказательство: Упражнение [Рекомендуется, ★★★].

Кроме того, Система F, подобно λ_{\rightarrow} , обладает свойством *нормализации* (normalization) — вычисление всякой правильно типизированной программы завершается.² В отличие от приведенных выше теорем о типовой безопасности, доказательство нормализации весьма сложно (в сущности, удивительно уже то, что это свойство вообще имеется, учитывая, что в чистом языке можно закодировать такие вещи, как функции сортировки (см. упражнение 23.4.12) без обращения к *fix*). Это доказательство, основанное на обобщении метода, представленного нами в главе 12, было одним из основных достижений докторской диссертации Жирара (Girard, 1972; см. также Girard, Lafont, and Taylor, 1989). С тех пор его метод доказательства был проанализирован и переработан многими другими исследователями (см. Gallier, 1990).

Теорема 23.5.3 [Нормализация]: *Правильно типизированные термы Системы F являются нормализующими.*

23.6. Стирание, типизируемость и реконструкция типов

Как и в случае λ_{\rightarrow} в §9.5, мы можем определить функцию *стирания типов* (type erasure), которая переводит термы Системы F в термы бестипового

²На самом деле, варианты Системы F с менее строгой операционной семантикой на основе полной бета-редукции обладают свойством *сильной нормализации* (strong normalization): гарантируется завершение любого пути нормализации, начинающегося с правильно типизированного терма.

лямбда исчисления, отбрасывая аннотации типов (включая все абстракции типов и применения типов):

$$\begin{aligned}
 erase(x) &= x \\
 erase(\lambda x:T_1. t_2) &= \lambda x. erase(t_2) \\
 erase(t_1 t_2) &= erase(t_1) erase(t_2) \\
 erase(\lambda x. t_2) &= erase(t_2) \\
 erase(t_1 [T_2]) &= erase(t_1)
 \end{aligned}$$

Терм m бестипового лямбда-исчисления называется *типизируемым* (typable) в Системе F, если существует какой-либо правильно типизированный терм t , такой, что $erase(t) = m$. Задача *реконструкции типов* (type reconstruction) состоит в том, чтобы, имея бестиповый терм m , сказать, можем ли мы отыскать некоторый правильно типизированный терм, дающий при стирании m .

Реконструкция типов для Системы F была одной из самых долгоживущих нерешенных проблем в области исследований по языкам программирования. Вопрос оставался открытым с начала 70-х до тех пор, пока Уэллс не ответил на него отрицательно в начале 90-х годов.

Теорема 23.6.1 [Уэллс (WELLS, 1994)]: *Для данного замкнутого терма m в бестиповом лямбда-исчислении задача о существовании правильно типизированного терма t Системы F, такого, что $erase(t) = m$, неразрешима.*

Известно, что не только реконструкция типов сама по себе, но и некоторые ее частичные формы в Системе F неразрешимы. Рассмотрим, например, следующее отношение «частичного стирания», которое сохраняет все аннотации типов, кроме (возможно) аргументов к применению типов. Заметим, что *местоположение* применений типов отмечается в частично стертых термах пустыми квадратными скобками.

$$\frac{x \lessdot x}{\lambda x:T_1. \quad t_2 \lessdot t'_2} \quad \frac{t_2 \lessdot t'_2}{\lambda x. t_2 \lessdot \lambda x:T_1. \quad t'_2} \quad \frac{\begin{array}{c} t_1 \lessdot t'_1 \\ t_2 \lessdot t'_2 \end{array}}{\begin{array}{c} t_1 t_2 \lessdot t'_1 t'_2 \\ t_1 \lessdot t'_1 \end{array}}$$

$$\frac{t_1 \lessdot t'_1 \quad t_2 \lessdot t'_2}{t_1 [T_2] \lessdot t'_1 [T_2]} \quad \frac{t_1 \lessdot t'_1 \quad t_2 \lessdot t'_2}{t_1 [T_2] \lessdot t'_1 []}$$

Теорема 23.6.2 [Пфенниг (PFENNING, 1993а)]: *Для данного терма s задача о существовании правильно типизированного терма t Системы F, такого, что $t \lessdot s$, неразрешима.*

Основываясь на более ранних наблюдениях Бёма (Boehm, 1985, 1989), Пфенниг показал, что такая форма реконструкции типов столь же сложна, как унификация высших порядков, про которую уже было известно, что она неразрешима. Интересно, что этот отрицательный результат прямо привел к созданию полезного метода частичной реконструкции типов (Pfenning, 1988, 1993а), основанного на ранних работах Юэ по эффективным полуалгоритмам для унификации высших порядков (Huet, 1975). Среди дальнейших улучшений в этом направлении исследований — более тонкий алгоритм для решения ограничений высших порядков (Dowek, Hardin, Kirchner, and Pfenning, 1996),

исключающий опасность незавершения алгоритма или порождения множественных решений. Использование родственных алгоритмов в таких языках, как LEAP (Pfenning and Lee, 1991), Elf (Pfenning, 1989) и FX (O'Toole and Gifford, 1989), показало, что на практике они ведут себя вполне прилично.

Другой подход к частичной реконструкции типов основан на сделанном Перри наблюдении о том, что экзистенциальные типы первого класса (см. главу 24) совместимы с используемым в ML механизмом `datatype` (Perry, 1990); эту идею дальше развили Лойфер и Одерский (Läufer, 1992; Läufer and Odersky, 1994). В сущности, конструкторы и деструкторы `datatype` можно рассматривать как явные аннотации типа, которые указывают места, где требуется упаковывать и распаковывать значения типов-непересекающихся объединений, где нужно сворачивать и разворачивать рекурсивные типы, а также, если добавлены экзистенциальные типы, где требуется вести их упаковку и распаковку. Реми (Rémy, 1994) удалось распространить эту идею на (импредикативные) универсальные типы первого класса. Недавнее предложение Одерского и Лойфера (Odersky and Läufer, 1996), которое развили далее Гарриг и Реми (Garrigue and Rémy, 1997), строит консервативное расширение реконструкции типов в стиле ML, позволяя программистам явно указывать типы аргументов функций, и эти аннотации (в отличие от выводимых автоматически) могут содержать универсальные кванторы. Таким образом, сужается разрыв между ML и более мощными импредикативными системами. Преимуществом этого семейства подходов является относительная простота и изящная интеграция с полиморфизмом языка ML.

Прагматический подход к частичной реконструкции типов для систем, одновременно включающих подтипы и импредикативный полиморфизм, называется *локальным выводом типов* (local type inference) (или *локальной реконструкцией типов*, local type reconstruction). Он был предложен Пирсом и Тёрнером (Pierce and Turner, 1998; см. также Pierce and Turner, 1997; Hosoya and Pierce, 1999). Локальный вывод типов присутствует также в некоторых современных языковых проектах, включая GJ (Bracha, Odersky, Stoutamire, and Wadler, 1998) и Funnel (Odersky and Zenger, 2001). В последнем вводится более мощная форма вывода, которая называется *окрашенным локальным выводом типов* (colored local type inference) (Odersky, Zenger, and Zenger, 2001).

Более простой, но менее предсказуемый алгоритм *жадного вывода типов* (greedy type inference) был предложен Карделли (Cardelli, 1993); подобные алгоритмы также используются в программах проверки доказательств для теорий с зависимыми типами, например, Nuprl (Howe, 1988) и Lego (Pollack, 1990). Здесь идея состоит в том, что любая аннотация типа может быть опущена программистом: для каждого такого случая процедура синтаксического разбора порождает новую переменную унификации X . При проверке типов алгоритму проверки подтипов можно задать вопрос, является ли некоторый тип S подтипов T , причем как S , так и T могут содержать переменные унификации. Проверка подтипов ведется как обычно до тех пор, пока не возникает подцель вида $X <: T$ или $T <: X$. В таком месте X конкретизируется типом T , и текущее ограничение удовлетворяется простейшим из возможных способов. Однако присваивание переменной X значения T может быть неоптимальным решением, и это может привести к дальнейшим неудачным проверкам под-

типирования для типов, включающих X , при том что другой выбор мог бы привести к успеху. Однако использование этого алгоритма на практике в реализации Карделли и в ранней версии языка Pict (Pierce and Turner, 2000) показывает, что жадный выбор алгоритма почти всегда правилен. Однако в случаях, когда выбор оказывается неверным, поведение жадного алгоритма может быть совершенно непонятным для программиста и приводить к таинственным сообщениям об ошибках далеко от того места, где была сделана неоптимальная конкретизация.

Упражнение 23.6.3 [★★★]: Из свойства нормализации следует, что бестиповы́й терм $\text{отега} = (\lambda x. x x) (\lambda y. y y)$ не может быть типизирован в Системе F , поскольку редукция отега никогда не достигает нормальной формы. Однако можно получить и более прямое, «комбинаторное» доказательство этого утверждения, рассматривая только правила, определяющие отношение типизации.

1. Назовем терм Системы F незащищенным (*exposed*), если это переменная, абстракция $\lambda x: T. t$ или применение $t s$ (т. е., если терм не является абстракцией типа $\lambda X. t$ или применением типа $t [S]$).

Покажем, что если терм t правильно типизирован (в некотором контексте) и $\text{erase}(t) = m$, то существует некоторый незащищенный терм s , такой, что $\text{erase}(s) = m$, и s правильно типизирован (возможно, в другом контексте).

2. Будем использовать запись $\lambda \bar{X}. t$, которая обозначает последовательность абстракций типа вида $\lambda X_1 \dots \lambda X_n. t$. Аналогично, будем писать $t [\bar{A}]$, обозначая таким образом вложенную последовательность применений типа $((t [A_1]) \dots [A_{n-1}]) [A_n]$ и $\forall \bar{X}. T$ для вложенной последовательности полиморфных типов $\forall X_1 \dots \forall X_n. T$. Следует учитывать, что эти последовательности могут быть пустыми. Например, если \bar{X} — пустая последовательность типовых переменных, то $\forall \bar{X}. T$ — просто T .

Покажем, что если $\text{erase}(t) = m$ и $\Gamma \vdash t : T$, то существует некоторый терм s вида $\lambda \bar{X}.$ (и $[\bar{A}]$) для некоторой последовательности типовых переменных \bar{X} , некоторой последовательности типов \bar{A} и некоторого незащищенного терма u , для которых $\text{erase}(s) = m$ и $\Gamma \vdash s : T$.

3. Покажем, что если t — незащищенный терм типа T (в контексте Γ) и $\text{erase}(t) = m n$, то t имеет вид $s u$ для некоторых термов s и u , таких, что $\text{erase}(s) = m$ и $\text{erase}(u) = n$, и при этом $\Gamma \vdash s : U \rightarrow T$ и $\Gamma \vdash u : U$.
4. Допустим, что $x: T \in \Gamma$. Покажем, что если $\Gamma \vdash u : U$ и $\text{erase}(u) = x x$, то либо

$$(a) \quad T = \forall \bar{X}. X_i, \text{ где } X_i \in \bar{X}, \text{ либо}$$

$$(b) \quad T = \forall \bar{X}_1 \bar{X}_2. T_1 \rightarrow T_2, \text{ где } [\bar{X}_1 \bar{X}_2 \mapsto A] T_1 = [\bar{X}_1 \mapsto B] (\forall \bar{Z}. T_1 \rightarrow T_2) \text{ для некоторых последовательностей типов } \bar{A} \text{ и } \bar{B}, \text{ таких, что } |\bar{A}| = |\bar{X}_1 \bar{X}_2| \text{ и } |\bar{B}| = |\bar{X}_1|.$$

5. Покажем, что, если $\text{erase}(s) = \lambda x.m$ и $\Gamma \vdash s : S$, то S имеет вид $\forall \bar{X}.S_1 \rightarrow S_2$ для некоторых \bar{X} , S_1 и S_2 .

6. Определяем самый левый лист (*leftmost leaf*) тела T :

$$\begin{aligned}\text{leftmost-leaf}(X) &= X \\ \text{leftmost-leaf}(S \rightarrow T) &= \text{leftmost-leaf}(S) \\ \text{leftmost-leaf}(\forall X.S) &= \text{leftmost-leaf}(S).\end{aligned}$$

Покажем, что, если $[\bar{X}_1 \bar{X}_2 \mapsto \bar{A}] (\forall \bar{Y}.T_1) = [\bar{X}_1 \mapsto \bar{B}] (\forall \bar{Z}. (\forall \bar{Y}.T_1) \rightarrow T_2)$, то должно выполняться $\text{leftmost-leaf}(T_1) = X_i$ для некоторого $X_i \in \bar{X}_1 \bar{X}_2$.

7. Покажем, что терм *отсек* не типизируем в Системе F.

23.7. Стирание и порядок вычислений

Операционная семантика Системы F, определенная на рис. 23.1 — это *семантика с передачей типов* (type-passing semantics): когда полиморфная функция принимает аргумент-тип, он действительно подставляется в тело функции. Реализация Системы F в главе 25 именно так и поступает.

В более практическом интерпретаторе или компиляторе для языка программирования, основанного на Системе F, такая манипуляция типами во время исполнения может приводить к существенным затратам. Более того, нетрудно заметить, что аннотации типов не играют никакой существенной роли во время исполнения, в том смысле, что при исполнении не принимается никаких решений на основе типов: можно взять правильно типизированную программу, произвольным образом переписать ее аннотации типов, и получить новую программу, которая ведёт себя точно так же. Поэтому многие полиморфные языки основаны на *семантике со стиранием типов* (type-erasure semantics), в которой после завершения фазы проверки типов вся типовая информация уничтожается, и уже получившиеся *бестиповые* термы интерпретируются либо компилируются в машинный код.³

Однако в полноценном языке программирования, в котором есть конструкции с побочными эффектами вроде изменяемых ячеек памяти или исключений, функция стирания типов требует более тщательного определения, чем функция полного стирания из §23.6. Например, если мы дополним Систему F примитивом `error` для сигнализации об ошибках (§14.1), то терм

```
let f = (λX.error) in 0;
```

дает при вычислении 0, поскольку $\lambda X.\text{error}$ — синтаксическое значение, и терм `error` в его теле не подлежит исполнению. Однако, получаемый при стирании терм

```
let f = error in 0;
```

³ В некоторых языках реализация с передачей типов диктуется наличием конструкций вроде *приведения типов* (casts) (§15.5). Как правило, высокопроизводительные реализации этих языков пытаются во время выполнения сохранять лишь минимальные остатки информации о типах, например, передавая типы только в полиморфные функции, в которых они действительно могут быть использованы.

при попытке его вычисления вызывает исключение.⁴ Это показывает, что абстракции типов все-таки *играют* существенную семантическую роль, предотвращая вычисление при стратегии вызова по значению, и, следовательно, могут отложить или предотвратить выполнение элементарных операций с побочными эффектами.

Это расхождение можно устраниТЬ, если ввести новую форму стирания, подходящую для вычисления с вызовом по значению. В новой форме абстракция типа дает при стирании *терм*-абстракцию.

$$\begin{aligned} \text{erase}_v(x) &= x \\ \text{erase}_v(\lambda x:T_1.t_2) &= \lambda x.\text{erase}_v(t_2) \\ \text{erase}_v(t_1 t_2) &= \text{erase}_v(t_1) \text{erase}_v(t_2) \\ \text{erase}_v(\lambda x.t_2) &= \lambda _. \text{erase}_v(t_2) \\ \text{erase}_v(t_1 [T_2]) &= \text{erase}_v(t_1) \text{dummyv} \end{aligned}$$

где `dummyv` — некоторое произвольное бестиповое значение, например, `unit`.⁵ Корректность новой функции стирания подтверждается наблюдением о том, что она может проводиться в любом порядке относительно бестипового вычисления.

Теорема 23.7.2 *Если $\text{erase}_v(t) = u$, то либо (1) как t , так и u являются нормальными формами относительно своих собственных отношений вычисления, либо (2) $t \rightarrow t'$, а $u \rightarrow u'$, причем $\text{erase}_v(t') = u'$.*

23.8. Варианты Системы F

Благодаря своему изяществу и мощи Система F занимает центральное место в теоретических исследованиях полиморфизма. Однако в разработке языков часто считается, что потеря реконструкции типов — слишком дорогая цена за гибкость, которая так редко используется в полную меру. Отсюда возникают несколько различных вариантов подмножеств Системы F, для которых задача реконструкции более решаема.

Самое известное из них — полиморфизм через `let` в стиле ML (§22.7), которое иногда называется *пренекс-полиморфизм* (prenex-polymorphism),

⁴Это связано с уже встречавшейся нам проблемой некорректного взаимодействия ссылок с полиморфизмом через `let` в стиле ML из §22.7. *Обобщение* (generalization) тела `let` в том примере здесь соответствует явной абстракции типа.

Упражнение 23.7.1 [★★]: Переведите опасный пример со с. 364 на язык Системы F с добавлением ссылок (рис. 13.1).

⁵В отличие от этого решения, *ограничение на значения* (value restriction), введенное нами в §22.7 для сохранения корректности реконструкции типов в стиле ML при наличии побочных эффектов, стирает абстракции типов: обобщение типовой переменной, в частности, противоположно стиранию абстракции типа. Корректность обеспечивается потому, что такие обобщения оказываются разрешены только там, где выводимая абстракция типа была бы непосредственно вложена в абстракцию терма или другой конструктор синтаксического значения, поскольку и то, и другое также останавливает вычисление.

поскольку его можно рассматривать как вариант Системы F, в котором типовые переменные имеют областью определения только типы без кванторов (*монотипы*, monotypes), а типы с кванторами (*политипы*, polytypes, или *схемы типов*, types schemas) не могут оказаться слева от стрелок. Особая роль конструкции `let` в ML затрудняет формальное выражение этого соответствия; подробности можно найти в Jim, 1995.

Еще одно хорошо изученное ограничение Системы F — *полиморфизм ранга 2* (rank-2 polymorphism), введенный Лейвантом (Leivant, 1983) и подробно исследованный многими другими (см. Jim, 1995, 1996). Тип называется типом ранга 2, если при его изображении в виде дерева никакой путь от его корня к квантору \forall не проходит слева от более чем двух стрелок. Например, $(\forall X. X \rightarrow X) \rightarrow \text{Nat}$ имеет ранг 2, а также $\text{Nat} \rightarrow \text{Nat}$ и $\text{Nat} \rightarrow (\forall X. X \rightarrow X) \rightarrow \text{Nat} \rightarrow \text{Nat}$. Однако, $((\forall X. X \rightarrow X) \rightarrow \text{Nat}) \rightarrow \text{Nat}$ типом ранга 2 не является. В системе ранга 2 все типы должны быть ранга 2. Эта система несколько более мощная, чем пренексный (ML) вариант, в том смысле, что она способна присвоить типы большему количеству лямбда-термов.

Кфури и Тюрин (Kfoury and Tiugun, 1990) доказали, что вычислительная сложность реконструкции типов для варианта Системы F ранга 2 равна сложности ML-фрагмента (т. е., DEXPTIME-полна). Кфури и Уэллс (Kfoury and Wells, 1999) построили первый правильный алгоритм реконструкции типов для системы ранга 2 и показали, что задача реконструкции типов для рангов 3 и выше Системы F неразрешима.

Ограничение ранга 2 может применяться и к другим мощным конструкторам типов помимо кванторов. Например, *типы-пересечения* (intersection types) (см. §15.7) можно ограничить рангом 2, если запретить типы, в которых оператор пересечения находится слева более чем от двух стрелок (Kfoury, Mairson, Turbak, and Wells, 1999). Варианты ранга 2 Системы F и системы типов первого порядка с пересечениями тесно связаны. Действительно, Джим (Jim, 1995) показал, что они могут присваивать типы в точности одним и тем же термам.

23.9. Параметричность

Вспомним, как в §23.4 мы определили тип `CBool` булевых значений, закодированных по Чёрчу:

```
CBool = ∀X. X → X → X;
```

и константы `tru` и `fls`:

```
tru = λX. λt:X. λf:X. t;
```

```
▷ tru : CBool
```

```
fls = λX. λt:X. λf:X. f;
```

```
▷ fls : CBool
```

Имея тип `CBool`, определения термов `tru` и `fls` можно выписать почти механически, просто глядя на структуру типа. Поскольку `CBool` начинается с

квантора \forall , любое его значение будет абстракцией типа, так что `tru` и `fls` обязаны начинаться с λX . Далее, поскольку тело `CBool` является функциональным типом $X \rightarrow X \rightarrow X$, каждое значение этого типа должно принимать два аргумента типа X — т. е. тела `tru` и `fls` обязаны начинаться с $\lambda t:X. \lambda f:X.$. Наконец, поскольку результирующий тип `CBool` — опять же, X , каждое значение типа `CBool` должно возвращать элемент типа X . Но поскольку X — параметр, единственны значения этого типа, которые мы в состоянии вернуть — это связанные переменные `t` и `f`; никаких других способов получить или создать значение этого типа у нас нет. Другими словами, `tru` и `fls`, в сущности, исчерпывают собой тип `CBool`. Строго говоря, `CBool` содержит некоторые другие термы, такие как $(\lambda b:CBool.b)$ `tru`, но интуитивно ясно, что каждый из них ведет себя либо как `tru`, либо как `fls`.

Это наблюдение является простым следствием мощного принципа *параметричности* (parametricity), который формализует единообразное поведение полиморфных программ. Параметричность была введена Рейнольдсом (Reynolds, 1974, 1983). В дальнейшем ее исследовали, наряду с родственными понятиями, Рейнольдс (Reynolds, 1984; Reynolds and Plotkin, 1993), Бейнбридж и др. (Bainbridge, Freyd, Scedrov, and Scott, 1990), Ма (Ma, 1992), Митчелл (Mitchell, 1986), Митчелл и Майер (Mitchell and Meyer, 1985), Хасегава (Hasegawa, 1991), Питтс (Pitts, 1987, 1989, 2000), Абади, Карделли, Куриен и Плоткин (Abadi, Cardelli, and Curien, 1993; Plotkin and Abadi, 1993; Plotkin, Abadi, and Cardelli, 1994), Уодлер (Wadler, 1989, 2001) и другие. Описание основ можно найти у Уодлера (Wadler, 1989).

23.10. Импредикативность

Полиморфизм Системы F часто называют *импредикативным* (impredicative). В общем случае, определение (множества, типа и т. п.) называют «импредикативным», если оно использует квантор, переменная которого может иметь значением сам определяемый объект. Например, в Системе F типовая переменная X в типе $T = \forall X. X \rightarrow X$ может иметь своим значением любой тип, включая и сам T (так что, например, можно конкретизировать тип T типом T и получить функцию из T в T). С другой стороны, полиморфизм в стиле ML часто называют *предикативным* (predicative) или *слоистым* stratified, поскольку значения типовых переменных ограничены монотипами, а те не содержат кванторов.

Термины «предикативный» и «импредикативный» происходят из логики. Куайн (Quine, 1987) проясняет их историю:

В переписке с Анри Пуанкаре ... Рассел предложил приписать парадокс [Рассела] тому, что он назвал заблуждением порочного круга. «Заблуждение» состояло в том, что класс определялся условием принадлежности, которое прямо или косвенно ссылалось на набор классов, среди которых находился и сам определяемый класс. Например, условие принадлежности, лежащее в основе парадокса Рассела, — отсутствие самовключения: x не является членом x . Парадокс происходит оттого, что переменной x в условии при-

надлежности разрешено быть, среди прочего, тем самым классом, который определяется условием принадлежности. Рассел и Пуанкаре стали называть такое условие принадлежности *импредикативным*, и дисквалифицировали его в качестве средства указания класса. Таким образом, парадоксы теории множеств, Расселовский и другие, были лишены силы [...].

Откуда же берутся термины «предикативный» и «импредикативный»? Рассел называет отброшенной банальностью о классах и условиях принадлежности к ним утверждение о том, что всякий предикат определяет класс; затем он приспособливается к отказу от этой банальности, отказывая в звании предиката таким условиям принадлежности, которые более не рассматриваются как определяющие классы. Таким образом, определение «предикативный» не относилось ни к конкретному иерархическому подходу, ни к метафоре последовательного построения; это было просто конкретное предложение Рассела и Пуанкаре о том, какие условия принадлежности можно считать продуктивными при образовании классов, или «предикативными». Однако вскоре ситуация перевернулась с ног на голову. Сегодня предикативная теория множеств — это конструктивная теория, а импредикативное определение понимается именно так, как объяснено в предыдущем абзаце, независимо от того, какие условия принадлежности мы предпочитаем рассматривать как определяющие классы.

23.11. Дополнительные замечания

Дополнительную библиографию по Системе F можно найти во вводной статье Рейнольдса ([Reynolds, 1990](#)) и в его же «Теориях языков программирования» ([«Theories of Programming Languages», Reynolds, 1998b](#)).

$\rightarrow \lambda$ На основе $\lambda \rightarrow$ (9.1)

Синтаксис

$t ::=$	термы:
x	переменная
$\lambda x:T.t$	абстракция
$t t$	применение
$\lambda X.t$	абстракция типа
$t [T]$	применение типа

$v ::=$	значения:
$\lambda x:T.t$	значение-абстракция
$\lambda X.t$	значение-абстракция типа

$T ::=$	типы:
X	типовая переменная
$T \rightarrow T$	тип функций
$\forall X.T$	универсальный тип

$\Gamma ::=$	контексты:
\emptyset	пустой контекст
$\Gamma, x:T$	связывание термовой переменной
Γ, X	связывание типовой переменной

Вычисление

 $t \rightarrow t'$

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \quad (\text{E-APP2})$$

$$(\lambda x:T_{11}.t_{12}) v_2 \rightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APPABS})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 [T_2] \rightarrow t'_1 [T_2]} \quad (\text{E-TAPP})$$

$$(\lambda X.t_{12}) [T_2] \rightarrow [X \mapsto T_2]t_{12} \quad (\text{E-TAPP TABS})$$

Типизация $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

$$\frac{\Gamma, x:T_1 \vdash t_2:T_2}{\Gamma \vdash \lambda x:T_1.t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \quad (\text{T-TABS})$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1 [T_2] : [X \mapsto T_2]X.T_{12}} \quad (\text{T-TAPP})$$

Рис. 23.1. Полиморфное лямбда-исчисление (Система F)

Глава 24

Экзистенциальные типы

Исследовав роль кванторов общности в системах типов (глава 23), естественно задаться вопросом, могут ли быть полезны в программировании кванторы *существования*. И действительно, кванторы существования оказываются изящной основой для абстракции данных и сокрытия информации.

24.1. Мотивация

Экзистенциальные типы концептуально не сложнее универсальных (на самом деле, в §24.3 мы увидим, что экзистенциальные типы без труда можно выразить в терминах универсальных). Однако формы для введения и устранения экзистенциальных типов синтаксически несколько тяжеловеснее, чем простые формы для абстракции и применения типов, связанные с универсальными типами, и иногда они поначалу кажутся непонятными. Полезно изложить интуитивные соображения, которые могут помочь побыстрее прокочить эту стадию.

Универсальные типы из главы 23 можно рассматривать с двух различных точек зрения. *Логическая интуиция* (logical intuition) говорит нам, что элемент типа $\forall X.T$ является значением, которое имеет тип $[X \mapsto S]T$ при любом выборе S . Это интуитивное представление соответствует взгляду на поведение со стиранием типов: например, полиморфная функция тождества $\lambda X.\lambda x:X.x$ при стирании дает бестиповую функцию тождества $\lambda x.x$, переводящую аргумент любого типа S в результат того же типа. Напротив, *операционная интуиция* (operational intuition) состоит в том, что элементом $\lambda X.T$ является функция, переводящая тип S в специализированный терм типа $[X \mapsto S]T$. Это интуитивное представление соответствует нашему определению Системы F из главы 23, согласно которому редукция применения типов считается отдельным шагом вычисления.

Аналогично, возможны две точки зрения на экзистенциальный тип, который записывается как $\{\exists X,T\}$. *Логическая интуиция* говорит, что элементом

По большей части в этой главе изучается Система F (рис. 23.1) с экзистенциальными типами (рис. 24.1). В примерах используются также записи (рис. 11.7) и числа (рис. 8.2). Соответствующая реализация на OCaml называется `fullpoly`.

$\{\exists X, T\}$ является значение типа $[X \mapsto S]T$ для *какого-то* типа S . С другой стороны, *операционная интуиция* говорит, что элементом $\{\exists X, T\}$ служит *пара*, которая записывается как $\{*_S, t\}$, состоящая из типа S и терма t , имеющего тип $[X \mapsto S]T$.

В этой главе мы будем особо подчеркивать операционный взгляд на экзистенциальные типы, поскольку он обеспечивает более близкую аналогию между экзистенциальными типами и модулями, с одной стороны, и абстрактными типами данных в языках программирования, с другой стороны. Наш конкретный синтаксис для экзистенциальных типов отражает эту аналогию: мы пишем $\{\exists X, T\}$ вместо более стандартной нотации $\exists X.T$ — фигурные скобки здесь подчеркивают, что значение экзистенциального типа является кортежом.

Чтобы понять экзистенциальные типы, нужно знать две вещи: как *конструировать* (или, в терминологии §9.4, *вводить*, introduce) элементы этих типов, и как эти значения *использовать* (или *устранять*, eliminate) в вычислениях.

Значение экзистенциального типа вводится путем построения пары из типа и терма. Этому действию соответствует запись $\{*_S, t\}$.¹ Полезное интуитивное объяснение — считать значение $\{*_S, t\}$ типа $\{\exists X, T\}$ простой формой *пакета* (package) или *модуля* (module) с одной (скрытой) компонентой-типов и одной компонентой-термом.² Тип S часто называют *скрытым типом представлением* (hidden representation type), или, иногда, чтобы подчеркнуть связь с логикой (см. §9.4), — *типом-свидетелем* (witness type) пакета. Например, пакет $p = \{*\text{Nat}, \{a=5, f=\lambda x:\text{Nat}. \text{succ}(x)\}\}$ имеет экзистенциальный тип $\{\exists X, \{a:X, f:X \rightarrow X\}\}$. Компонента-тип p равняется Nat , а компонентой-значением служит запись, содержащая поле a типа X и поле f типа $X \rightarrow X$, для некоторого X (а именно Nat).

Тот же пакет p *также* имеет тип $\{\exists X, \{a:X, f:X \rightarrow \text{Nat}\}\}$, поскольку его правая компонента является записью с полями a и f типов X и $X \rightarrow \text{Nat}$, для некоторого X (а именно Nat). Этот пример показывает, что в общем случае программа проверки типов не может принять автоматическое решение о том, какому экзистенциальному типу принадлежит данный пакет: программист должен указывать, какой тип имеется в виду. Простейший способ добиться этого — просто добавить к каждому пакету аннотацию с явным указанием его типа. Таким образом, полная форма введения для экзистенциальных значений будет выглядеть так:

¹ Мы помечаем компоненту-тип такой пары значком $*$, чтобы четче отделить ее от обычновенных кортежей-термов (§11.7). Еще один распространенный способ записи для введения экзистенциального значения выглядит так: pack $X=S$ with t .

² Разумеется, нетрудно обобщить модули такого вида, добавив возможность указания нескольких компонент-типов или компонент-термов, но мы пока оставим по одной компоненте каждого вида, чтобы нотация оставалась обозримой. Эффекта нескольких компонент-типов можно добиться вложением нескольких экзистенциальных типов с одной типовой переменной, а эффект нескольких компонент-термов достигается, если в качестве правой компоненты использовать кортеж или запись:

$$\{*_S, *_S, t_1, t_2\} \stackrel{\text{def}}{=} \{*_S, \{*_S, \{t_1, t_2\}\}\}$$

```
p = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X → X}};
▷ p : {∃X, {a:X, f:X → X}}
```

или (тот же пакет с другим типом):

```
p1 = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X → Nat}};
```

```
▷ p1 : {∃X, {a:X, f:X → Nat}}
```

Аннотация типа, вводимая ключевым словом `as`, аналогична оператору присыпывания типа, введенному в §11.4. Присыпывание позволяет пометить *любой* терм типом, который он по идеи должен иметь. В сущности, мы добавляем одно присыпывание к конкретному синтаксису конструкции создания пакета. Правило типизации для введения экзистенциального квантора таково:

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{ *U, t_2 \} \text{ as } \{ \exists X, T_2 \} : \{ \exists X, T_2 \}} \quad (\text{T-PACK})$$

Следует заметить, что, согласно этому правилу, пакеты с *различными* типами представления могут принадлежать *одному и тому же* экзистенциальному типу. Например:

```
p2 = {*Nat, 0} as {∃X, X};
▷ p2 : {∃X, X}
p3 = {*Bool, true} as {∃X, X};
▷ p3 : {∃X, X}
```

Или, в более полезном варианте,

```
p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X → Nat}};
▷ p4 : {∃X, {a:X, f:X → Nat}}
p5 = {*Bool, {a=true, f=λx:Bool. 0}} as {∃X, {a:X, f:X → Nat}};
▷ p5 : {∃X, {a:X, f:X → Nat}}
```

Упражнение 24.1.1 [★]: *Вот еще три вариации на ту же тему:*

```
p6 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X → X}};
▷ p6 : {∃X, {a:X, f:X → X}}
```

```
p7 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:Nat → X}};
▷ p7 : {∃X, {a:X, f:Nat → X}}
```

```

▷ p7 : {∃X, {a:X, f:Nat → X}}
p8 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:Nat →
Nat}};
▷ p8 : {∃X, {a:Nat, f:Nat → Nat}}
Чем они менее полезны, чем p4 и p5?

```

Аналогия с модулями также помогает нам понять конструкцию устранения экзистенциального квантора. Если экзистенциальный пакет соответствует модулю, то устранение пакета подобно директиве `open` или `import`: оно позволяет использовать содержимое модуля в каком-то другом месте программы, но сохраняет абстрактную природу типовой компоненты модуля. Этого можно добиться связыванием через своего рода сопоставление с образцом:

$$\frac{\Gamma \vdash t_1 : \{ \exists X, T_{12} \} \quad \Gamma, X, x:T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2} \quad (\text{T-UNPACK})$$

То есть, если t_1 — выражение, дающее при вычислении экзистенциальный пакет, то мы можем связать его типовую и термовую компоненты с переменными образца X и x и использовать эти переменные при вычислении t_2 . (Часто также встречается такой конкретный синтаксис для устранения экзистенциального квантора: `open t1 as {X, x} in t2`.)

Возьмем, например, определенный выше пакет $p4$ типа $\{ \exists X, \{a:X, f:X \rightarrow \text{Nat}\} \}$. Выражение устраниния

```

let {X, x}=p4 in (x.f x.a);

▷ 1 : Nat

```

открывает $p4$ и использует поля его тела ($x.f$ и $x.a$) при вычислении числового результата. В теле формы устраниния можно также упоминать типовую переменную X :

```

let {X, x}=p4 in (λy:X. x.f y) x.a;

▷ 1 : Nat

```

То, что тип представления пакета при проверке типов в теле остается абстрактным, означает, что над x разрешены только те операции, которые обеспечивает ее «абстрактный тип» $\{a:X, f:X \rightarrow \text{Nat}\}$. В частности, мы не можем использовать $x.a$ как число:

```

let {X, x}=p4 in succ(x.a);

▷ Ошибка: аргумент succ не является числом

```

Это ограничение совершенно справедливо, поскольку, как мы видели выше, пакет с тем же самым экзистенциальным типом, что у $p4$, может использовать в качестве типа представления не только `Nat`, но и `Bool` (а также любой другой тип).

Есть еще один, менее очевидный, способ не пройти проверку типов для конструкции устранения экзистенциального квантора. В правиле T-UNPACK типовая переменная X присутствует в контексте, в котором вычисляется тип t_2 , но *отсутствует* в контексте заключения правила. Это означает, что тип результата T_2 не может содержать свободных вхождений X , поскольку иначе в заключении эти вхождения X окажутся вне области видимости.

```
let {X,x}=p in x.a;
```

▷ Ошибка: нарушение области видимости!

Этот вопрос обсуждается подробнее в §25.5.

Правило вычисления для экзистенциальной конструкции не представляет сложности:

```
let {X,x}={*T11,v12} as T1) in t2 → [X ↦ T11][x ↦ v12]t2  
(E-UNPACKPACK)
```

Если первое подвыражение `let` сведено к конкретному пакету, то мы можем подставить компоненты этого пакета вместо переменных X и x в теле t_2 . В терминах аналогии с модулями, это правило можно рассматривать как шаг *компоновки* (linking), когда символические имена (X и x), указывающие на компоненты отдельно скомпилированного модуля, заменяются реальным содержимым этого модуля.

Поскольку в этом правиле все вхождения типовой переменной X подвергаются подстановке, получающаяся программа имеет доступ к конкретному содержимому пакета. Это еще один пример явления, которое мы уже несколько раз наблюдали: по мере вычисления выражения могут становиться «более типизированными» — в частности, неправильно типизированное выражение может перейти в правильно типизированное.

Правила, расширяющие Систему F экзистенциальными типами, приведены на рис. 24.1.

24.2. Абстракция данных при помощи экзистенциальных типов

Во вводной главе (§1.2) мы говорили, что полезность систем типов далеко не исчерпывается обнаружением локальных ошибок программирования вроде `2 + true`: типы также оказывают весомую поддержку при создании больших программ. В частности, типы помогают соблюдать не только абстракции, встроенные в язык, но и *абстракции, определенные программистом* (programmer-defined abstractions), — т. е. они защищают не только машину от программы, но и части программы друг от друга.³ В этом разделе рассматриваются два различных стиля абстракции — классические *абстрактные типы*

³Справедливости ради, нужно сказать, что типы — не единственный механизм для защиты абстракций, определяемых программистом. В бестиповых языках подобного эффекта можно достигнуть при помощи замыканий, объектов или специализированных конструкций вроде *единиц компиляции* (units) в MzScheme (Flatt and Felleisen, 1998).

$\rightarrow \forall \exists$

Расширяет Систему F (23.1)

Новые синтаксические формы

$t ::= \dots$ термы:
 $\{*T, t\} \text{ as } T$ упаковка
 $\text{let } \{X, x\} = t \text{ in } t$ распаковка
 ka

$v ::= \dots$ значения:
 $\{*T, v\} \text{ as } T$ значение-пакет

$T ::= \dots$ типы:
 $\{\exists X, T\}$ экзистенциальный тип

Новые правила вычисления

 $t \rightarrow t'$

$\text{let } \{X, x\} = (\{*T_{11}, v_{12}\} \text{ as } T_1) \text{ in } t_2$
 $\rightarrow [X \mapsto T_{11}] [x \mapsto v_{12}] t_2$

(E-UNPACKPACK)

$$\frac{t_{12} \rightarrow t'_{12}}{\{*T_{11}, t_{12}\} \text{ as } T_1} \quad (\text{E-PACK})$$

$$\rightarrow \{*T_{11}, t'_{12}\} \text{ as } T_1$$

$$\frac{t_1 \rightarrow t'_1}{\begin{aligned} &\text{let } \{X, x\} = t_1 \text{ in } t_2 \\ &\rightarrow \text{let } \{X, x\} = t'_1 \text{ in } t_2 \end{aligned}} \quad (\text{E-UNPACK})$$

Новые правила типизации $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U] T_2}{\begin{aligned} &\Gamma \vdash \{*U, t_2\} \text{ as } \{\exists X, T_2\} \\ &: \{\exists X, T_2\} \end{aligned}} \quad (\text{T-PACK})$$

$$\frac{\begin{aligned} &\Gamma \vdash t_1 : \{\exists X, T_{12}\} \\ &\Gamma, X, x : T_{12} \vdash t_2 : T_2 \end{aligned}}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2} \quad (\text{T-UNPACK})$$

Рис. 24.1. Экзистенциальные типы

данных (abstract data types) и объекты (objects). Основой для обсуждения служат экзистенциальные типы.

В отличие от кодирования объектов, рассмотренного нами в главе 18, все примеры в этой главе являются чисто функциональными (purely functional) программами. Этот выбор сделан нами исключительно ради удобства обсуждения: механизмы для обеспечения модульности и абстракции почти совершенно независимы от того, хранят ли определяемые абстракции изменяющее состояние или нет. (В упражнениях 24.2.2 и 24.2.3 этот момент подчеркнут. В них разрабатываются императивные варианты некоторых чисто функциональных примеров из основного текста.) Мы предпочитаем чисто функциональные примеры по двум причинам: а) такой выбор ведет к тому, что наши примеры определяются в рамках более простой и экономной формальной системы, и б) при работе с чисто функциональными программами возникающие проблемы типизации иногда оказываются более интересными (а их решения, соответственно, более показательными). Это получается потому, что при императивном программировании изменяемые переменные могут служить «по-

бочным каналом связи», через который происходит прямое общение удаленных друг от друга частей программы. В чисто функциональных программах вся информация передается между различными частями программы в виде аргументов или результатов функций, и таким образом, «видима» для системы типов. Это в особенности верно в случае объектов, и из-за этого нам придется отложить работу с некоторыми важными возможностями языка (с подтипами и наследованием реализаций) до главы 32, в которой в нашем распоряжении будут некоторые ещё более мощные механизмы теории типов.

Абстрактные типы данных

Традиционный *абстрактный тип данных*, или *АТД* (abstract data type, ADT) состоит из а) имени типа *A*, б) типа конкретного представления *T*, в) реализаций некоторых операций по созданию, извлечению информации и манипулированию значениями типа *T* и г) *барьера абстракции* (abstraction boundary), отделяющего представление и операции от внешнего мира. Внутри барьера абстракции элементы типа рассматриваются как конкретные значения (типа *T*). Вне барьера они рассматриваются абстрактно, как значения типа *A*. Значения типа *A* можно передавать из процедуры в процедуру, хранить в структурах данных и т. д., но нельзя напрямую обращаться к их представлению и изменять его — над *A* разрешены только те операции, которые предоставляет АТД.

Вот, например, объявление абстрактного типа данных чисто функциональных счетчиков, записанное на псевдокоде, близком к языкам Ada (U.S. Dept. of Defense, 1980) или Clu (Liskov et al., 1981).

```
ADT counter =
  type Counter
  representation Nat
  signature
    new : Counter,
    get : Counter → Nat,
    inc : Counter → Counter;
  operations
    new = 1,
    get = λi:Nat. i,
    inc = λi:Nat. succ(i);

  counter.get (counter.inc counter.new);
```

Имя абстрактного типа — *Counter*; его конкретным представлением является *Nat*. Реализации операций работают с объектами *Counter* на конкретном уровне, как со значениями типа *Nat*: *new* — это просто константа 1; операция *inc* представляет собой функцию-последователь; *get* — это функция тождества. Раздел *signature* указывает, как эти операции должны использоваться извне; при этом некоторые экземпляры *Nat* в их конкретных типах заменяются на *Counter*. Барьер абстракции огораживает участок от ключевого слова *ADT* до закрывающей точки с запятой. В оставшейся части программы (т. е. в последней ее строчке) связь между *Counter* и *Nat* разорвана, так что един-

ственное, что можно сделать с константой `counter.new` — это использовать ее как аргумент для `counter.get` или `counter.inc`.

Этот псевдокод почти посимвольно можно перевести в наше исчисление с экзистенциальными типами. Сначала мы создаем экзистенциальный пакет, содержащий внутренности АТД:

```
counterADT =
  {*Nat,
   {new = 1,
    get = λi:Nat. i,
    inc = λi:Nat. succ(i)}}
  as {∃Counter,
      {new: Counter,
       get: Counter → Nat,
       inc: Counter → Counter}};
```

▷ counterADT : {∃Counter,
 {new:Counter, get:Counter → Nat, inc:Counter →
 Counter}}

Затем мы открываем пакет, вводя типовую переменную `Counter`, указывающую на скрытый тип представления пакета и термовую переменную `counter` для доступа к операциям:

```
let {Counter, counter} = counterADT in
  counter.get (counter.inc counter.new);

▷ 2 : Nat
```

Версия с использованием экзистенциальных типов читается несколько труднее, чем псевдокод, содержащий много синтаксического сахара, однако структура двух этих фрагментов кода идентична.

В общем случае тело конструкции `let`, открывающей экзистенциальный пакет, содержит всю оставшуюся часть программы:

```
let {Counter, counter} = <пакет counter> in
  <остаток программы>
```

В этой оставшейся части тип `Counter` может использоваться точно так же, как базовые типы, встроенные в язык. Можно определять функции, работающие со счетчиками:

```
let {Counter, counter}=counterADT in
let add3 = λc:Counter. counter.inc (counter.inc (counter.inc c)) in
  counter.get (add3 counter.new);

▷ 4 : Nat
```

Можно даже определять новые абстрактные типы, представление которых включает в себя счетчики. Например, следующая программа определяет АТД двухпозиционных переключателей, используя счетчик в качестве (не слишком эффективного) типа представления:

```
let {Counter.counter} = counterADT in
```

```

let {FlipFlop, flipflop} =
  {*Counter,
   {new      = counter.new,
    read     = λc:Counter. iseven (counter.get c),
    toggle   = λc:Counter. counter.inc c,
    reset    = λc:Counter. counter.new}}
  as {∃FlipFlop,
      {new:     FlipFlop, read:  FlipFlop → Bool,
       toggle: FlipFlop → FlipFlop, reset: FlipFlop →
FlipFlop}} in

flipflop.read (flipflop.toggle (flipflop.toggle flipflop.new));

▷ false: Bool

```

Таким образом можно разбить большую программу на некоторое количество объявлений АТД, каждый из которых использует типы и операции, предоставляемые предыдущими АТД, чтобы построить свои типы и операции и упаковать их для следующих АТД в виде ясно и четко определенной абстракции.

Ключевое свойство сокрытия информации в таком стиле — *независимость представлений* (representation independence). Можно подставить другую реализацию АТД счетчика — скажем, такую, в которой в качестве внутреннего представления используется запись с полем типа `Nat`, а не просто `Nat`,

```

counterADT =
  {*{x:Nat},
   {new = {x=1},
    get = λi:{x:Nat}. i.x,
    inc = λi:{x:Nat}. {x=succ(i.x)}}
  as {∃Counter,
      {new:Counter, get: Counter → Nat, inc:Counter →
Counter}}};

▷ counterADT : {∃Counter,
                  {new:Counter, get:Counter → Nat, inc:Counter →
Counter}}

```

с полной уверенностью в том, что вся программа останется правильно типизированной, поскольку мы знаем наверняка, что оставшаяся часть программы никак не может обратиться к экземплярам `Counter` иначе, как через `get` и `inc`.

Опыт показывает, что стиль программирования, основанный на абстрактных типах данных, может принести громадные улучшения в надежности и легкости сопровождения больших систем. Тому есть несколько причин. Во-первых, такой стиль ограничивает область видимости изменений в программе. Как мы только что видели, можно заменить одну реализацию АТД другой, с возможным изменением как типа представления, так и реализации операций, никак не влияя на остальную программу, поскольку правила типизации для экзистенциальных пакетов обеспечивают независимость остальной программы

мы от внутреннего представления АТД. Во-вторых, такой стиль побуждает программиста ограничивать зависимости между частями программы, делая сигнатуры АТД как можно меньше. Наконец, и может быть, это самое важное, этот стиль заставляет программистов думать о *проектировании абстракций* (designing abstractions).

Упражнение 24.2.1 [РЕКОМЕНДУЕТСЯ, ★ ★ ★]: Используя приведенный пример в качестве образца, определите абстрактный тип стеков (*stacks*) чисел с операциями *push*, *push* (поместить значение на стек), *top* (вернуть значение на вершине стека), *pop* (вернуть новый стек с удаленной вершиной) и *isempty*. Используйте в качестве представления тип *List*, введенный в упражнении 23.4.3. Напишите простую программу, которая создает стек, записывает в него два числа и получает верхний элемент стека. Лучше всего выполнять это упражнение в интерактивном режиме. Запустите интерпретатор *fullomega* и скопируйте содержимое файла *test.f* (где определяется конструктор типов *List* и связанные с ним операции) в начало собственного файла.

Упражнение 24.2.2 [РЕКОМЕНДУЕТСЯ, ★★]: Постройте АТД изменяемых счетчиков, используя ссылочные ячейки, определенные в главе 13. Пусть операция *push* будет не константой, а функцией, которая принимает аргумент типа *Unit* и возвращает счетчик, а возвращаемый тип операции *inc* пусть будет не *Counter*, а *Unit*. (Интерпретатор *fullomega* поддерживает как экзистенциальные типы, так и ссылки.)

Экзистенциальные объекты

Схема «упаковать, а затем открыть», которую мы видели в предыдущем подразделе, типична для программирования с использованием экзистенциальных пакетов в качестве АТД. Пакет определяет абстрактный тип и связанные с ним операции, а мы открываем каждый пакет немедленно после его создания, связывая при этом типовую переменную, обозначающую абстрактный тип, и открывая абстрактный доступ к операциям над АТД. В этом подразделе мы увидим, как простая разновидность *объектно-ориентированной* (object-oriented) абстракции данных может быть представлена в виде другой идиомы программирования, основанной на экзистенциальных типах. Эта объектная модель далее разрабатывается в главе 32.

Мы снова будем использовать в качестве основного примера простые счетчики, как в случае с экзистенциальными АТД, а также в наших предыдущих опытах с объектами в главах 18 и 19. Мы снова выбираем чисто функциональный стиль, так что посылка сообщения *inc* объекту не изменяет его внутреннего состояния, а возвращает *новый* счетчик с увеличенным внутренним состоянием.

Объект-счетчик состоит из двух основных компонент: числа, являющегося внутренним состоянием счетчика, и пары методов, *get* и *inc*, используемых для работы с этим состоянием. Нам также нужно добиться того, чтобы единственным способом узнать состояние или изменить его был бы вызов одного из этих двух методов. Этого можно добиться, если завернуть состояние и методы

в экзистенциальный пакет и абстрагироваться от типа состояния. Например, объект-счетчик, имеющий значение 5, можно записать так:

```
c = {*Nat,
      {state = 5,
       methods = {get = λx:Nat. x,
                  inc = λx:Nat. succ(x)}}
     as Counter;
```

где

```
Counter = {ΞX, {state:X, methods: {get:X → Nat, inc:X → X}}};
```

Чтобы вызвать метод объекта-счетчика, мы открываем экзистенциальный пакет и применяем соответствующий элемент его поля `methods` к полю `state`. Например, чтобы получить текущее значение `c`, мы пишем:

```
let {X,body} = c in body.methods.get(body.state);  
▷ 5 : Nat
```

Обобщая, можно написать маленькую функцию, которая «посыпает сообщение `get`» любому счетчику:

```
sendget = λc:Counter.  
         let {X,body} = c in  
           body.methods.get(body.state);  
  
▷ sendget : Counter → Nat
```

Вызов метода `inc` для объекта-счетчика выглядит несколько сложнее. Если мы просто повторим код для `get`, программа проверки типов выдаст ошибку:

```
let {X,body} = c in body.methods.inc(body.state);  
▷ Ошибка: нарушение области видимости!
```

поскольку типовая переменная `X` оказывается несвязанной в типе тела `let`. В самом деле, написанный нами код интуитивного смысла тоже не имеет, поскольку результатом метода `inc` является голое внутреннее состояние, а не объект. Чтобы удовлетворить одновременно процедуру проверки типов и свои неформальные понятия о том, к чему должен приводить вызов `inc`, мы должны взять новое внутреннее состояние и упаковать его в виде объекта-счетчика, используя при этом ту же запись с методами и тот же тип внутреннего состояния, что и в исходном объекте:

```
c1 = let {X,body} = c in  
      {*X,  
       {state = body.methods.inc(body.state),  
        methods = body.methods}}  
     as Counter;
```

В общем случае, чтобы «послать счетчику сообщение `inc`», можно написать

```
sendinc = λc:Counter.  
          let {X,body} = c in
```

```

{*X,
  {state = body.methods.inc(body.state),
   methods = body.methods}}
as Counter;

▷ sendinc : Counter → Counter

```

Более сложные операции над счетчиками можно построить на основе этих двух базовых операций:

```

add3 = λc:Counter. sendinc (sendinc (sendinc c));

▷ add3 : Counter → Counter

```

Упражнение 24.2.3 [РЕКОМЕНДУЕТСЯ, ★ ★ ★]: Реализуйте переключатели *FlipFlop* как объекты, используя в качестве типа внутреннего представления объекты *Counter*. В качестве образца используйте АТД *FlipFlop* из предыдущего подраздела.

Упражнение 24.2.4 [РЕКОМЕНДУЕТСЯ, ★★]: При помощи интерпретатора *fullomega* реализуйте императивный вариант объектов *Counter* по образцу упражнения 24.2.2.

Сравнение объектов и АТД

Примеры из предыдущего раздела не составляют полноценной модели объектно-ориентированного программирования. Многие возможности, рассмотренные нами в главах 18 и 19 — подтипы, классы, наследование, рекурсия через *self* и *super*, здесь отсутствуют. Мы вернемся к их моделированию в главе 32, когда наша система типов обогатится некоторыми существенными чертами. Однако уже сейчас мы можем провести интересное сравнение наших простых объектов с обсуждавшимися ранее АТД.

При самом поверхностном взгляде эти две схемы программирования диаметрально противоположны: в процессе программирования с помощью АТД пакеты открываются немедленно после создания; когда же пакеты используются для построения объектов, они остаются закрытыми как можно дольше — до момента, когда их *приходится* открывать, чтобы применить один из методов к внутреннему состоянию объекта.

Вследствие этого различия «абстрактный тип счетчиков» в этих двух стилях означает разные вещи. В программе, написанной в стиле с АТД, значение счетчиков, с которыми работает клиентский код вроде функции *add3*, являются элементами внутреннего типа представления (т. е., обычновенными числами). В объектной программе каждый счетчик представляет собой отдельный пакет — содержащий не только число, но и представления методов *get* и *inc*. Это стилистическое различие отражается в том, что в случае с АТД тип *Counter* является связанный переменной, которая вводится в конструкции *let*, а в объектном случае *Counter* обозначает весь экзистенциальный тип

```
{*X, {state:X, methods:{get:X → Nat, inc:X → X}}}
```

Таким образом, во время исполнения все значения счетчиков, порожденные АТД — это просто неупакованные элементы одного и того же типа представления, и существует единственная реализация операций со счетчиками, которая работает с этим внутренним представлением. Напротив, каждый объект-счетчик несет свой тип представления и свой собственный набор методов, работающих с этим типом представления.

Эти расхождения между АТД и объектами приводят к различным практическим достоинствам и недостаткам. Одно очевидное отличие состоит в том, что, поскольку каждый объект сам выбирает свое представление и несет свои собственные операции, в одной программе можно работать с несколькими представлениями одного и того же типа объектов. Это особенно удобно, когда в системе присутствуют подтипы и наследование: можно определить один общий тип объектов, а затем породить множество уточнений, каждое из которых обладает своим несколько (или совершенно) отличающимся внутренним представлением. Поскольку все экземпляры этих уточненных классов имеют один и тот же общий тип, с ними может работать один общий код, их можно совместно хранить в списках и т. п.

Например, библиотека пользовательского интерфейса может определить общий класс `Window` (окно) с подклассами `TextWindow` (текстовое окно), `ContainerWindow` (окно-контейнер), `ScrollableWindow` (окно с полосами прокрутки), `TitledWindow` (окно с заголовком), `DialogBox` (диалоговое окно) и т. д. В каждом из этих подклассов будут собственные переменные экземпляра (например, `TextWindow` может содержать переменную типа `String`, представляющую текущее содержимое окна, а `ContainerWindow` может хранить список объектов типа `Window`), и у него будут свои собственные реализации для таких операций, как `repaint` (перерисовка) и `handleMouseEvent` (обработка события от мыши). С другой стороны, определение типа `Window` как АТД порождает менее гибкую структуру. Конкретный тип представления `Window` должен будет содержать вариантный тип ([§11.10](#)) с вариантом для каждой разновидности окна, и к этому варианту будут приписаны данные, относящиеся именно к этой разновидности. Такие операции, как `repaint`, будут содержать `case` по вариантам и выполнять соответствующий специализированный код. Если разновидностей окон существует много, то такое монолитное объявление абстрактного типа `Window` может стать слишком большим и запутанным.

Второе важное практическое различие между объектами и АТД касается статуса *бинарных операций* (binary operations) — операций, которым требуется два или более аргумента одного и того же абстрактного типа. Чтобы обсуждение этого вопроса было понятнее, требуется различать два вида бинарных операций:

- Некоторые бинарные операции можно реализовать целиком в терминах общедоступных операций над двумя абстрактными значениями. Например, чтобы реализовать проверку счетчиков на равенство, нужно только получить у каждого из них информацию о его текущем состоянии (с помощью `get`) и сравнить два полученных числа — т. е. операция `equal` с тем же успехом может существовать за пределами барьера абстрации, защищающего конкретное представление счетчиков. Такие операции мы будем называть *слабыми бинарными операциями* (weak binary

operations).

- Другие бинарные операции невозможно реализовать, не имея конкретного привилегированного доступа к обоим абстрактным значениям. Например, пусть мы реализуем абстракцию, представляющую собой множество чисел. После изучения нескольких учебников по алгоритмам мы выбираем конкретное представление множеств в виде помеченных деревьев, сохраняющих некоторый сложный инвариант. Эффективная реализация операции `union` (объединение) для двух множеств должна рассматривать их конкретно как деревья. Однако мы *не хотим* давать доступ к конкретному представлению множеств в нашем общедоступном интерфейсе абстракции множества. Таким образом, нужно будет устроить привилегированный доступ операции `union` к обоим ее аргументам, запрещенный для обычного кода — т. е. операция `union` должна существовать внутри барьера абстракции. Такие операции мы будем называть *сильными бинарными операциями* (strong binary operations).

Слабые бинарные операции легко реализуются в обоих рассматриваемых на-
ми стилях, поскольку для них несущественно, помещаем ли мы их внутри или
вне барьера абстракции. Если мы поместим их снаружи, их можно определить
просто как самостоятельные функции (которые принимают по необходимости
либо объекты, либо значения АТД). Точно так же можно поместить их внутрь
АТД (тогда у них будет конкретный доступ к представлению аргументов, хотя
он им и не нужен). Поместить такую операцию внутрь объекта лишь нена-
много сложнее, поскольку теперь тип объекта оказывается рекурсивным.⁴

```
EqCounter = {ΞX, {state:X, methods:{get:X → Nat, inc:X → X,  
eq:X → EqCounter →  
Bool}}}
```

Однако сильные бинарные операции в нашей модели невозможно представить как методы объектов. Мы можем выразить их *типы* так же, как мы это делали для слабых бинарных методов:

```
NatSet = {ΞX, {state:X, methods:{empty:X, singleton:Nat → X,  
member:X → Nat → Bool,  
union:X → NatSet → X}}}
```

Однако не существует удовлетворительного способа *реализации* объекта этого типа: все, что нам известно о втором аргументе операции `union` — это что он предоставляет операции типа `NatSet`, однако они не дают никакого способа выяснить, каковы элементы множества, и поэтому вычислить их объединение невозможно.

Упражнение 24.2.5 [★]: Почему нельзя использовать следующий тип?

```
NatSet = {ΞX, {state:X, methods:{empty:X, singleton:Nat → X,  
member:X → Nat → Bool,  
union:X → X → X}}}
```

⁴Такая разновидность рекурсии нетривиальным образом взаимодействует с наследованием реализаций, см. Bruce, Cardelli, Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Leavens, and Pierce, 1996.

Итак, единые представления АТД прямо поддерживают бинарные операции, а множественные представления объектов не предоставляют этой поддержки, но дают взамен полезную гибкость. Преимущества двух стилей дополняют друг друга; ни один из них не является очевидно лучшим.

К этому обсуждению нужно сделать одну поправку. Приведенные сравнения относятся к простой, «чистой» модели объектов, описанной в этой главе. Классы в популярных объектно-ориентированных языках программирования, таких как C++ или Java, специально спроектированы так, что *позволяют* определять некоторые разновидности сильных бинарных методов. Можно сказать, что они обеспечивают некоторый компромисс между чистыми объектами и чистыми АТД, рассмотренными нами в этой главе. В этих языках типом объекта в точности является имя класса, который этот объект породил, и этот тип считается отличным от имен всех остальных классов, даже если они предоставляют в точности те же операции (ср. §19.3). А именно, каждый тип объектов в этих языках имеет *единственную* реализацию, данную в соответствующем объявлении класса. Более того, подклассы в этих языках могут добавлять переменные экземпляра только к тем переменным экземпляра, которые унаследованы от надклассов. Следствием этих ограничений является то, что объект, принадлежащий типу С, всегда обладает всеми переменными экземпляра, упомянутыми в (единственном) объявлении класса С (и возможно, некоторыми другими). Поэтому для метода такого класса имеет смысл принимать другой С в качестве аргумента и непосредственно обращаться ко всем его переменным экземпляра, если он использует только те переменные, которые определены в С. Поэтому сильные бинарные операции вроде `union` можно только определять в качестве методов. Такого рода «гибридные» объектные модели формально описаны в работах Пирса и Тёрнера (Pierce and Turner, 1993) и Катияра и др. (Katiyar, Luckham, and Mitchell, 1994). Более подробно их изучали Фишер и Митчелл (Fisher and Mitchell, 1996, 1998; Fisher, 1996b,a).

24.3. Кодирование экзистенциальных типов

Кодирование пар в виде полиморфного типа в §23.4 подсказывает нам, что аналогичным образом экзистенциальные типы можно закодировать в виде универсальных типов, поскольку интуитивно элемент экзистенциального типа представляет собой пару из типа и значения:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y. (\forall X. T \rightarrow Y) \rightarrow Y.$$

То есть, экзистенциальный пакет рассматривается как значение, которое, получая тип результата и *продолжение* (continuation), вызывает продолжение и получает окончательный результат. Продолжение принимает два аргумента — тип X и значение типа T, — и использует их при вычислении окончательного результата.

Такое представление экзистенциальных типов однозначно определяет кодирование конструкций упаковки и распаковки. Чтобы закодировать пакет

$$\{*, t\} \text{ as } \{\exists X, T\}$$

мы должны с помощью S и t построить значение типа $\forall Y. (\forall X. T \rightarrow Y) \rightarrow Y$. Тип начинается с универсального квантора, а телом его является функциональный тип. Следовательно, элемент этого типа должен начинаться с двух абстракций:

$$\{*S, t\} \text{ as } \{\exists X, T\} \stackrel{\text{def}}{=} \lambda Y. \lambda f: (\forall X. T \rightarrow Y) \dots$$

Чтобы закончить построение, нужно вернуть результат типа Y ; ясно, что получить его можно, только применив функцию f к подходящим аргументам. Сначала мы даем ей тип S (это естественный выбор, поскольку других типов в нашем распоряжении сейчас нет):

$$\{*S, t\} \text{ as } \{\exists X, T\} \stackrel{\text{def}}{=} \lambda Y. \lambda f: (\forall X. T \rightarrow Y). f [S] \dots$$

Применение типа $f [S]$ имеет тип $[X \mapsto S](T \rightarrow Y)$, т. е., $([X \mapsto S]T) \rightarrow Y$. В качестве второго аргумента мы даем ему терм t , который, согласно правилу T-PACK, имеет тип $[X \mapsto S]T$:

$$\{*S, t\} \text{ as } \{\exists X, T\} \stackrel{\text{def}}{=} \lambda Y. \lambda f: (\forall X. T \rightarrow Y). f [S] t$$

Типом всего терма-применения $f [S] t$ будет Y , как и требуется.

Чтобы закодировать конструкцию распаковки `let {x, x}=t1 in t2`, мы рассуждаем аналогично. Во-первых, правило типизации T-UNPACK говорит нам, что терм t_1 должен иметь тип вида $\{\exists X, T_{11}\}$, что t_2 должен иметь тип T_2 (в расширенном контексте, включающем связывания X и $x:T_{11}$), и что в качестве типа результата всего выражения `let...in...` ожидается тип T_2 ⁵. Как и при кодировании по Чёрчу в §23.4, интуитивная идея здесь состоит в том, что форма введения ($\{*S, t\}$) кодируется в виде активного значения, которое «само производит собственное устранение». Таким образом, кодирование формы устраниния здесь просто должно взять экзистенциальный пакет t_1 и применить его к достаточному количеству аргументов, чтобы получить результат желаемого типа T_2 :

$$\text{let } \{X, x\}=t_1 \text{ in } t_2 \stackrel{\text{def}}{=} t_1 \dots$$

Первым аргументом t_1 должен быть требуемый тип результата для всего выражения, т. е., T_2 :

$$\text{let } \{X, x\}=t_1 \text{ in } t_2 \stackrel{\text{def}}{=} t_1 [T_2] \dots$$

Тип терма-применения $t_1 [T_2]$ равен $(\forall X. T_{11} \rightarrow T_2) \rightarrow T_2$. Значит, если сейчас нам удастся передать ему аргумент типа $(\forall X. T_{11} \rightarrow T_2)$, задача будет выполнена. Такой аргумент можно получить путем *абстракции* тела t_2 по переменным X и x :

$$\text{let } \{X, x\}=t_1 \text{ in } t_2 \stackrel{\text{def}}{=} t_1 [T_2] (\lambda X. \lambda x: T_{11}. t_2)$$

Кодирование завершено.

⁵Строго говоря, то, что перевод требует некоторой информации, отсутствующей в синтаксисе термов, означает, что на самом деле мы занимаемся переводом *деревьев вывода типов* (typing derivations), а не термов. С подобной ситуацией мы уже встречались в определении *семантики на основе преобразования типов* (coercion semantics) для подтиповирования в §15.6.

Упражнение 24.3.1 [РЕКОМЕНДУЕТСЯ, ★★ →]: Возьмите чистый лист бумаги и, не заглядывая в книгу, восстановите кодирование, приведенное выше.

Упражнение 24.3.2 [★★★]: Какие утверждения необходимо доказать, чтобы убедиться в том, что кодирование экзистенциальных типов выполнено правильно?

Упражнение 24.3.3 [★★★★]: Можно ли провести операцию в обратном направлении и закодировать универсальные типы в терминах типов экзистенциальных?

24.4. Дополнительные замечания

Соответствие между АТД и экзистенциальными типами впервые было описано Митчеллом и Плоткиным (Mitchell and Plotkin, 1988); они же заметили и связь с объектами. Пирс и Тёрнер (Pierce and Turner, 1994) подробно исследовали эту связь — см. главу 32, в которой приведены подробности и дальнейшие ссылки. Различные преимущества объектов и АТД обсуждали Рейнольдс (Reynolds, 1975), Кук (Cook, 1991), Брюс и др. (Bruce, Cardelli, Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Leavens, and Pierce, 1996) и многие другие. В частности, работа Брюса и др. (Bruce, Cardelli, Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Leavens, and Pierce, 1996) содержит подробное обсуждение бинарных методов.

Мы убедились, что экзистенциальные типы служат в теории типов естественным основанием для простой формы абстрактных типов данных. Для адекватного отображения (близкой, но намного более богатой) проблематики систем модулей (module systems), существующих в языках вроде ML, были исследованы многие более сложные механизмы. Начать изучение литературы в этой области можно с работ Карделли и Лероя (Cardelli and Leroy, 1990), Лероя (Leroy, 1994), Харпера и Лилибридж (Harper and Lillibridge, 1994), Лилибриджа (Lillibridge, 1997), Харпера и Стоуна (Harper and Stone, 2000) и Крэри и др. (Crary, Harper, and Dreyer, 2002).

Структура типов представляет собой синтаксическую дисциплину для обеспечения уровней абстракции.

Джон Рейнольдс
(Reynolds, 1983)

Глава 25

Реализация Системы F на ML

Теперь расширим реализацию λ_{\rightarrow} из гл. 10 и включим в нее универсальные и экзистенциальные типы из гл. 23 и 24. Поскольку правила, определяющие эту систему, управляются синтаксисом (подобно самой λ_{\rightarrow}), реализация ее на OCaml не представляет труда. Наиболее интересное расширение по сравнению с интерпретатором λ_{\rightarrow} — представление типов, которые могут содержать связывания переменных (в кванторах). Для этих типов мы используем метод *индексов de Брауна* (de Bruijn indices), описанный в гл. 6.

25.1. Представление типов без использования имен

Для начала расширим синтаксис типов типовыми переменными, а также кванторами общности и существования.

```
type ty =
  TyVar of int * int
  | TyArr of ty * ty
  | TyAll of string * ty
  | TySome of string * ty
```

Здесь действуют такие же соглашения, как при представлении термов в §7.1. Типовые переменные состоят из двух целых чисел: первое указывает расстояние до конструкции, связывающей переменную, а второе, используемое для проверки целостности, равно ожидаемому размеру контекста. Кванторы хранят имя связываемой ими переменной, которое используется функциями распечатки.

Затем расширим контексты, позволяя им хранить связывания для типовых переменных помимо термовых. Для этого мы добавляем новый конструктор в тип `binding`:

```
type binding =
  NameBind
  | VarBind of ty
  | TyVarBind
```

Как и в предыдущих реализациях, конструкция связывания `NameBind` используется только функциями синтаксического анализа и распечатки. Конструктор `VarBind`, как и раньше, хранит в себе тип. Новый конструктор `TyVarBind` никаких дополнительных данных не несет, поскольку (в отличие от термовых переменных) типовые переменные в этой системе не несут никаких дополнительных ограничений. В системе с ограниченной квантификацией (глава 26) и видами высшего порядка (глава 29) с каждым `TyVarBind` будет связана соответствующая аннотация.

25.2. Сдвиг типов и подстановка

Так как теперь типы содержат переменные, требуется определить функции для сдвига и подстановки типов.

Упражнение 25.2.1 [★]: Используя в качестве образца функцию сдвига термов из определения 6.2.1 (с. 97), выпишите математическое определение аналогичной функции, сдвигающей переменные в типах.

В §7.2 мы демонстрировали сдвиг и подстановку для термов в виде двух отдельных функций, но отметили, что реализация, доступная на сайте книги, использует единую функцию «отображения» для обеих задач. Можно определить аналогичную функцию отображения также и для сдвига и подстановки типов. Рассмотрим теперь эти функции.

Основное наблюдение состоит в том, что сдвиг и подстановка ведут себя совершенно одинаково со всеми конструкторами, кроме переменных. Если мы абстрагируем их поведение на переменных, они совпадают. Вот, например, специализированная функция сдвига для типов, которую мы получаем, механически переводя на OCaml решение упражнения 25.2.1:

```
let typeShiftAbove d c tyT =
  let rec walk c tyT = match tyT with
    TyVar(x,n) → if x >= c then TyVar(x+d,n+d) else TyVar(x,n+d)
    | TyArr(tyT1,tyT2) → TyArr(walk c tyT1,walk c tyT2)
    | TyAll(tyX,tyT2) → TyAll(tyX,walk (c+1) tyT2)
    | TySome(tyX,tyT2) → TySome(tyX,walk (c+1) tyT2)
  in walk c tyT
```

Среди аргументов для этой функции — величина `d`, на которую требуется сдвинуть свободные переменные; предельная величина (отсечка) `c`, ниже которой сдвигать *не следует* (чтобы избежать сдвига переменных, связанных кванторами внутри типа); а также подлежащий сдвигу тип `tyT`.

Если мы абстрагируем обработку варианта `TyVar` из `typeShiftAbove` в новый аргумент `onvar`, а также избавимся от аргумента `d`, который упоминается только в варианте `TyVar`, то мы получим обобщенную функцию отображения:

```
let tymap onvar c tyT =
  let rec walk c tyT = match tyT with
    TyArr(tyT1,tyT2) → TyArr(walk c tyT1,walk c tyT2)
  | TyVar(x,n) → onvar c x n
  | TyAll(tyX,tyT2) → TyAll(tyX,walk (c+1) tyT2)
  | TySome(tyX,tyT2) → TySome(tyX,walk (c+1) tyT2)
  in walk c tyT
```

из которой можно восстановить функцию сдвига, снабдив `tymap` вариантом для `TyVar` в качестве параметра (функции, абстрагированной по `c`, `x` и `n`):

```
let typeShiftAbove d c tyT =
  tymap
    (fun c x n → if x >= c then TyVar(x+d,n+d) else TyVar(x,n+d))
    c tyT
```

Удобно также определить специализированную версию `typeShiftAbove` для случая, когда исходная отсечка равна 0:

```
let typeShift d tyT = typeShiftAbove d 0 tyT
```

При помощи другой конкретизации `tymap` можно также реализовать операцию подстановки типа `tyS` вместо типовой переменной номер `j` в типе `tyT`:

```
let typeSubst tyS j tyT =
  tymap
    (fun j x n → if x=j then (typeShift j tyS) else (TyVar(x,n)))
    j tyT
```

Когда мы будем при проверке типов и вычислении использовать подстановку типов, подстановка всегда будет производиться вместо нулевой (самой внешней) переменной, и нам всегда нужно будет сдвинуть результат, чтобы эта переменная исчезла. Для этого используется вспомогательная функция `typeSubstTop`.

```
let typeSubstTop tyS tyT =
  typeShift (-1) (typeSubst (typeShift 1 tyS) 0 tyT)
```

25.3. Термы

На уровне термов требуется проделать аналогичную работу. Сначала расширим тип данных `term` из главы 10 формами введения и устраниния универсальных и экзистенциальных типов:

```
type term =
  TmVar of info * int * int
  | TmAbs of info * string * ty * term
  | TmApp of info * term * term
  | TmTAbs of info * string * term
  | TmTApp of info * term * ty
  | TmPack of info * ty * term * ty
  | TmUnpack of info * string * string * term * term
```

Определения сдвига и подстановки для термов похожи на те, что мы видели в главе 10. Однако здесь мы выпишем их через общую функцию отображения, как в предыдущем разделе мы сделали для типов. Функция отображения выглядит так:

```
let tmmap onvar ontype c t =
  let rec walk c t = match t with
    TmVar(fi,x,n) → onvar fi c x n
  | TmAbs(fi,x,tyT1,t2) → TmAbs(fi,x,ontype c tyT1,walk (c+1) t2)
  | TmApp(fi,t1,t2) → TmApp(fi,walk c t1,walk c t2)
  | TmTAbs(fi,tyX,t2) → TmTAbs(fi,tyX,walk (c+1) t2)
  | TmTApp(fi,t1,tyT2) → TmTApp(fi,walk c t1,ontype c tyT2)
  | TmPack(fi,tyT1,t2,tyT3) →
      TmPack(fi,ontype c tyT1,walk c t2,ontype c tyT3)
  | TmUnpack(fi,tyX,x,t1,t2) →
      TmUnpack(fi,tyX,x,walk c t1,walk (c+2) t2)
  in walk c t
```

Заметим, что `tmmap` принимает *четыре* аргумента — на один больше, чем `tymap`. Чтобы понять, почему это так, обратите внимание на то, что термы могут содержать два различных типа переменных: термовые переменные и типовые переменные, встроенные в аннотации типа внутри термов. Поэтому, например, при сдвиге есть два типа «листьев», которые нам надо обработать: термовые переменные и типы. Параметр `ontype` сообщает функции отображения, что следует делать при обработке конструктора термов, содержащего аннотацию типа, как, например, в случае `TmAbs`. Если бы у нас был язык большего размера, было бы еще несколько таких случаев.

Сдвиг термов можно определить, передав соответствующие аргументы в `tmmap`.

```
let termShiftAbove d c t =
  tmmap
    (fun fi c x n → if x >= c then TmVar(fi,x+d,n+d)
     else TmVar(fi,x,n+d))
    (typeShiftAbove d)
    c
  t

let termShift d t = termShiftAbove d 0 t
```

Для термовых переменных мы проверяем отсечку и создаем новую переменную точно так же, как это было сделано в `typeShiftAbove`. Для типов — вызываем функцию сдвига для типов, определенную в предыдущем разделе.

Функция для подстановки одного терма вместо другого выглядит аналогично:

```
let termSubst j s t =
  tmmap
    (fun fi j x n → if x=j then termShift j s else TmVar(fi,x,n))
    (fun j tyT → tyT)
    j
  t
```

Заметим, что аннотации типов не изменяются в `termSubst` (типы не могут содержать термовых переменных, так что подстановка термов на них никак не влияет).

Нам также нужна функция для подстановки *типа* в *терм* — она используется, например, в правиле вычисления для применения типа:

$$(\lambda X.t_{12}) [T_2] \rightarrow [X \mapsto T_2]t_{12} \quad (\text{E-TAPP TABS})$$

Ее также можно определить через функцию отображения термов:

```
let rec tytermSubst tyS j t =
  tmmap (fun fi c x n → TmVar(fi, x, n))
    (fun j tyT → typeSubst tyS j tyT) j t
```

В этом случае функция, которую мы передаем в `tmmap` для обработки термовых переменных, — это функция тождества (она просто восстанавливает исходную термовую переменную); когда мы видим аннотацию типа, мы проводим в ней подстановку на уровне типов.

Наконец, как мы это делали для типов, определим вспомогательные функции, которые упаковывают базовые функции подстановки для использования в `eval` и `typeof`.

```
let termSubstTop s t =
  termShift (-1) (termSubst 0 (termShift 1 s) t)
let tytermSubstTop tyS t =
  termShift (-1) (tytermSubst (typeShift 1 tyS) 0 t)
```

25.4. Вычисление

Расширения функции `eval` прямо порождаются на основе правил вычисления по рис. 23.1 и 24.1. Вся сложная работа уже проделана в функциях подстановки из предыдущего раздела.

```
let rec eval1 ctx t = match t with
  ...
  | TmTApp(fi, TmTAbs(_, x, t11), tyT2) →
    tytermSubstTop tyT2 t11
  | TmTApp(fi, t1, tyT2) →
    let t1' = eval1 ctx t1 in
    TmTApp(fi, t1', tyT2)
  | TmUnpack(fi, _, _, TmPack(_, tyT11, v12, _), t2) when isval ctx v12 →
    tytermSubstTop tyT11 (termSubstTop (termShift 1 v12) t2)
  | TmUnpack(fi, tyX, x, t1, t2) →
    let t1' = eval1 ctx t1 in
    TmUnpack(fi, tyX, x, t1', t2)
  | TmPack(fi, tyT1, t2, tyT3) →
    let t2' = eval1 ctx t2 in
    TmPack(fi, tyT1, t2', tyT3)
  ...
  ...
```

Упражнение 25.4.1 [★]: Почему в первом варианте `TmUnpack` требуется `termShift`?

25.5. Типизация

Новые варианты в функции `typeof` также прямо следуют из правил типизации для абстракции и применения типов, а также для упаковки и распаковки экзистенциальных пакетов. Мы приводим полное определение `typeof`, чтобы новые варианты для `TmTAbs` и `TmTApp` можно было сравнить со старыми, для обычной абстракции и обычного применения функции.

```
let rec typeof ctx t =
  match t with
  | TmVar(fi,i,_) → getTypeFromContext fi ctx i
  | TmAbs(fi,x,tyT1,t2) →
    let ctx' = addbinding ctx x (VarBind(tyT1)) in
    let tyT2 = typeof ctx' t2 in
    TyArr(tyT1, typeShift (-1) tyT2)
  | TmApp(fi,t1,t2) →
    let tyT1 = typeof ctx t1 in
    let tyT2 = typeof ctx t2 in
    (match tyT1 with
     | TyArr(tyT11,tyT12) →
       if (=) tyT2 tyT11 then tyT12
       else error fi "некоcтoвcтвие типа параметра"
     | _ → error fi "ожидается функциональный тип")
  | TmTAbs(fi,tyX,t2) →
    let ctx = addbinding ctx tyX TyVarBind in
    let tyT2 = typeof ctx t2 in
    TyAll(tyX,tyT2)
  | TmTApp(fi,t1,tyT2) →
    let tyT1 = typeof ctx t1 in
    (match tyT1 with
     | TyAll(_,tyT12) → typeSubstTop tyT2 tyT12
     | _ → error fi "ожидается универсальный тип")
  | TmPack(fi,tyT1,t2,tyT) →
    (match tyT with
     | TySome(tyY,tyT2) →
       let tyU = typeof ctx t2 in
       let tyU' = typeSubstTop tyT1 tyT2 in
       if (=) tyU tyU' then tyT
       else error fi "некoсtovcтvие объявленному типу"
     | _ → error fi "ожидается экзистенциальный тип")
  | TmUnpack(fi,tyX,x,t1,t2) →
    let tyT1 = typeof ctx t1 in
    (match tyT1 with
     | TySome(tyY,tyT11) →
       let ctx' = addbinding ctx tyX TyVarBind in
       let ctx'' = addbinding ctx' x (VarBind tyT11) in
       let tyT2 = typeof ctx'' t2 in
       typeShift (-2) tyT2
     | _ → error fi "ожидается экзистенциальный тип")
```

Наиболее интересный участок кода — случай `TmUnpack`. Здесь мы: 1) проверяем тип подвыражения `t1` и убеждаемся, что оно имеет экзистенциаль-

ный тип $\{\exists X, T_1\}$; 2) расширяем контекст Γ связыванием типовой переменной X и связыванием термовой переменной $x:T_1$, а также убеждаемся, что t_2 имеет некоторый тип T_2 ; 3) сдвигаем индексы свободных переменных в T_2 вниз на два так, чтобы этот тип имел смысл относительно исходного контекста Γ ; 4) возвращаем получившийся тип в качестве типа всего выражения `let...in...`.

Ясно, что если X имеет свободные вхождения в T_2 , то на третьем шаге мы получим бессмыленный тип, содержащий свободные переменные с отрицательными индексами; в этом месте проверка типов должна потерпеть неудачу. Мы можем этого добиться, если переопределим функцию `typeShiftAbove` так, чтобы она замечала попытки создания типовой переменной с отрицательным индексом и сообщала об ошибке, а не возвращала бессмыслицу.

```
let typeShiftAbove d c tyT =
  t umap
    (fun c x n → if x >= c then
      if x+d < 0 then err "Неверная область видимости!"
      else TyVar(x+d, n+d)
      else TyVar(x, n+d))
  c tyT
```

Такая проверка будет сообщать об ошибке области видимости всякий раз, когда тип, вычисляемый нами для тела t_2 конструкции устранения экзистенциального типа `let {X,X}=t1 in t2`, содержит связанную типовую переменную X .

```
let {X,x} = (*Nat,0) as {\exists X,X}) in x;
> Ошибка: Неверная область видимости!
```


Глава 26

Ограниченнaя квантификациa

Многие интересные проблемы в языках программирования проявляются при взаимодействии конструкций, которые относительно просты, если рассматривать их по отдельности. В этой главе мы вводим *ограниченную квантификацию* (bounded quantification), возникающую при сочетании полиморфизма и подтипов. При этом существенно увеличивается как выразительная сила системы, так и ее метатеоретическая сложность. Исчисление, с которым мы будем работать, называется F_{\leq} (произносится « F sub»). Оно играло центральную роль в исследованиях по языкам программирования с момента открытия в середине 80-х; в частности, в работах по основаниям объектно-ориентированного программирования.

26.1. Мотивация

Простейший способ совместить подтипы и полиморфизм состоит в том, чтобы считать их ортогональными характеристиками языка, т. е. рассмотреть систему, по существу, являющуюся объединением систем из глав 15 и 23. Такое сочетание не вызывает теоретических трудностей и может быть полезно по тем же причинам, по которым подтипы и полиморфизм полезны отдельно друг от друга. Однако как только мы получаем обе характеристики в одном языке, возникает соблазн сочетать их более интересным способом. В качестве иллюстрации рассмотрим простой пример — в дальнейшем мы встретим другие примеры в §26.3, а также разберем ещё более практические случаи в главах 27 и 32.

Предположим, что f является функцией тождества на записях с числовым полем a :

По большей части в этой главе исследуется чистая Система F_{\leq} (рис. 26.1). В примерах используются также записи (11.7) и числа (8.2). Соответствующие реализации на OCaml называются `fullsub` и `fullfomsub`. (Для большинства примеров достаточно интерпретатора `fullsub`; для примеров, включающих сокращения типов с параметрами, такие как `Pair`, требуется `fullfomsub`.)

```
f = λx:{a:Nat}. x;
▷ f : {a:Nat} → {a:Nat}
```

Если `ra` — запись, имеющая такое поле,

```
ra = {a=0};
```

то мы можем применить `f` к `ra` — в любой из рассмотренных нами ранее систем, — и получить запись того же типа.

```
f ra;
▷ {a=0} : {a:Nat}
```

Аналогично, если мы определим запись `rab` с двумя полями, `a` и `b`,

```
rab = {a=0, b=true};
```

то мы также можем применить `f` к `rab`, применяя правило включения (T-SUB, рис. 15.1) и расширяя тип `rab` до `{a:Nat}`, чтобы она соответствовала типу, ожидаемому функцией `f`.

```
f rab;
▷ {a=0, b=true} : {a:Nat}
```

Однако тип результата при таком применении имеет только поле `a`. Следовательно, такой терм, как `(f rab).b`, окажется неправильно типизированным. Другими словами, пропустив `rab` через функцию тождества, мы потеряли возможность обращаться к ее полю `b`!

Используя полиморфизм Системы F, мы можем записать `f` иначе:

```
fpoly = λX. λx:X. x;
▷ fpoly : ∀X. X → X
```

Применение функции `fpoly` к `rab` (с соответствующим аргументом-типовом) дает желаемый результат:

```
fpoly [{a:Nat, b:Boolean}] rab;
▷ {a=0, b=true} : {a:Nat, b:Boolean}
```

Однако, превратив тип `x` в переменную, мы потеряли некоторую потенциальную полезную информацию. Допустим, например, что нужно написать другую версию `f`, которая возвращает пару из исходного аргумента и значения его поля `a`, увеличенного на единицу:

```
f2 = λx:{a:Nat}. {orig=x, asucc=succ(x.a)};
▷ f2 : {a:Nat} → {orig:{a:Nat}, asucc:Nat}
```

С помощью подтипования мы по-прежнему можем применить `f2` как к `ra`, так и к `rab`, во втором случае теряя поле `b`.

```
f2 ra;
> {orig={a=0}, asucc=1} : {orig:{a:Nat}, asucc:Nat}

f2 rab;
> {orig={a=0,b=true}, asucc=1} : {orig:{a:Nat}, asucc:Nat}
```

Но теперь полиморфизм никак не может нам помочь. Если мы, как и раньше, заменим тип x переменной X , мы потеряли ограничение, согласно которому x должен быть записью с полем a , а это ограничение нужно, чтобы вычислить поле $asucc$ результата.

```
f2poly = λX. λx:X. {orig=x, asucc=succ(x.a)};
> Ошибка: Ожидается тип записей
```

Информация об операционном поведении $f2$, которую мы хотели бы выразить в ее типе, состоит в том, что $f2$ принимает аргумент любого типа записей R , в которых есть числовое поле a , и возвращает как результат запись, содержащую поле типа R и поле типа Nat . С помощью отношения подтипов мы можем выразить это кратко: $f2$ принимает аргумент любого подтипа R типа $\{a:\text{Nat}\}$ и возвращает запись с полем типа R и полем типа Nat . Эту интуитивную идею можно формализовать, введя *ограничение на подтипы* (subtyping constraint) для связанной переменной X в $f2poly$.

```
f2poly = λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)};
> f2poly : ∀X<:{a:Nat}. X → {orig:X, asucc:Nat}
```

Эта так называемая *ограниченная квантификация* (bounded quantification) является основной характеристикой Системы $F_{<:}$.

26.2. Определения

С формальной точки зрения, Система $F_{<:}$ получается сочетанием типов и термов Системы F из главы 23 с отношением подтипов из главы 15, а также добавлением к кванторам общности ограничений на подтипы. Аналогично можно определить и ограниченные кванторы существования, как мы увидим в §26.5.

На самом деле, есть даже два разумных способа определения отношения подтипов в $F_{<:}$. Они отличаются формулировкой правила для сравнения ограниченных кванторов (*S-All*): одна версия, так называемое *ядерное* (kernel) правило легче описывается формально, но обладает меньшей гибкостью, а *полное* (full) правило подтипов более выразительно, но вызывает технические трудности. В следующих разделах мы подробно опишем обе версии, сначала вводя ядерный вариант, а затем, в §26.2, полный вариант. Когда нам нужно будет точно указать, о каком варианте идет речь, мы будем называть версии исчисления соответственно *ядерной* $F_{<:}$ (kernel $F_{<:}$) и *полной* $F_{<:}$ (full $F_{<:}$). Использование $F_{<:}$ без прилагательных относится к обеим системам.

На рис. 26.1 приведено полное определение ядерной $F_{<:}$; отличия от предыдущих систем выделены.

$\rightarrow \forall <: \text{Top}$ Основана на Системе F (23.1) и простом подтиповании
(15.1)

Синтаксис $t ::=$ x $\lambda x:T.t$ $t t$ $\lambda x <: T . t$ $t [T]$	термы: переменная абстракция применение абстракция типа применение типа
---	---

$v ::=$ $\lambda x:T.t$ $\lambda x <: T . t$	значения: значение-абстракция значение-абстракция типа
--	---

$T ::=$ X Top $T \rightarrow T$ $\forall X <: T . T$	типы: типовая переменная максимальный тип тип функций универсальный тип
---	--

$\Gamma ::=$ \emptyset $\Gamma, x:T$ $\Gamma, X <: T$	контексты: пустой контекст связывание термовой переменной связывание типовой переменной
--	---

Вычисление	$t \rightarrow t'$
$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2}$	(E-APP1)
$\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2}$	(E-APP2)
$\frac{t_1 \rightarrow t'_1}{t_1 \ [T_2] \rightarrow t'_1 \ [T_2]}$	(E-TAPP)

$$(\lambda x <: T_{11} . t_{12}) \ [T_2] \rightarrow [x \mapsto T_2] t_{12} \quad (\text{T-TAPP TABS})$$

$$(\lambda x:T_{11} . t_{12}) \ v_2 \rightarrow [x \mapsto v_2] t_{12} \quad (\text{E-APPABS})$$

Подтипы	$\boxed{\Gamma \vdash S <: T}$
----------------	--------------------------------

$$\boxed{\Gamma \vdash S <: S} \quad (\text{S-REFL})$$

$$\boxed{\Gamma \vdash S <: U \quad \Gamma \vdash U <: T} \quad \boxed{\Gamma \vdash S <: T} \quad (\text{S-TRANS})$$

$$\boxed{\Gamma \vdash S <: \text{Top}} \quad (\text{S-TOP})$$

$$\boxed{\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T}} \quad (\text{S-TVAR})$$

$$\boxed{\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}} \quad (\text{S-ARROW})$$

$$\boxed{\frac{\Gamma, X <: U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: U_1 . S_2 <: \forall X <: U_1 . T_2}} \quad (\text{S-ALL})$$

Типизация	$\boxed{\Gamma \vdash t : T}$
------------------	-------------------------------

$$\boxed{\frac{x : T \in \Gamma}{\Gamma \vdash x : T}} \quad (\text{T-VAR})$$

$$\boxed{\frac{\Gamma, x:T_1 \vdash t_2:T_2}{\Gamma \vdash \lambda x:T_1 . t_2 : T_1 \rightarrow T_2}} \quad (\text{T-ABS})$$

$$\boxed{\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}}} \quad (\text{T-APP})$$

$$\boxed{\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1 . t_2 : \forall X <: T_1 . T_2}} \quad (\text{T-TABS})$$

$$\Gamma \vdash t_1 : \forall X <: T_{11} . T_{12} \quad \Gamma \vdash t_2 <: T_{11}$$

Ограниченнaя и неограниченaя квантификация

Из этого определения сразу видно одно обстоятельство: синтаксис $F_{<}$ разрешает *только* ограниченную квантификацию: обычная, неограниченная квантификация Системы F исчезла. Это потому, что она не нужна: ограниченный квантор с ограничением Тор охватывает все подтипы Тор — а значит, все типы вообще. Таким образом, мы восстанавливаем неограниченную квантификацию с помощью такого сокращенного правила:

$$\forall X. T \stackrel{\text{def}}{=} \forall X <: \text{Top}. T$$

Это сокращение часто используется в дальнейшем.

Области видимости

Важная техническая подробность, неочевидная из рис. 26.1, касается областей видимости типовых переменных. Очевидно, что всякий раз, когда мы используем утверждение типизации вида $\Gamma \vdash t : T$, мы хотим, чтобы свободные типовые переменные, встречающиеся в t и T , были определены в Γ . Но что происходит со свободными типовыми переменными, которые входят в типы *внутри* Γ ? В частности, какие из следующих контекстов должны считаться корректными с точки зрения областей видимости?

$$\begin{aligned}\Gamma_1 &= X <: \text{Top}, y : X \rightarrow \text{Nat} \\ \Gamma_2 &= y : X \rightarrow \text{Nat}, X <: \text{Top} \\ \Gamma_3 &= X <: \{a : \text{Nat}, b : X\} \\ \Gamma_4 &= X <: \{a : \text{Nat}, b : Y\}, Y <: \{c : \text{Bool}, d : X\}\end{aligned}$$

Контекст Γ_1 , несомненно, корректен: он вводит типовую переменную X , а затем — термовую переменную y , в типе которой упоминается X . Терм, который может привести к такому контексту в процессе проверки типов, будет иметь вид $\lambda X <: \text{Top}. \lambda y : X \rightarrow \text{Nat}. t$. Ясно, что переменная X в типе y связана вышележащей λ . С другой стороны, по тем же причинам контекст Γ_2 должен быть неверным, поскольку в термах, которые могли бы привести к такому контексту — скажем, для терма $\lambda y : X \rightarrow \text{Nat}. \lambda X <: \text{Top}. t$ неясно, какова ожидаемая область видимости X .

Случай контекста Γ_3 более интересен. Можно доказывать, что он *правилен* и возникает в термах вроде $\lambda X <: \{a : \text{Nat}, b : X\}. t$, в которых второе вхождение X связано. Для этого нужно только считать, что область видимости связывания X включает свою собственную верхнюю границу (а также, как обычно, все, что стоит справа от связывания). Разновидность ограниченной квантификации, содержащая это уточнение, называется *F-ограниченной квантификацией* (*F*-bounded quantification) (Canning, Cook, Hill, Olthoff, and Mitchell, 1989b). *F*-ограниченная квантификация часто упоминается в обсуждении типов в объектно-ориентированном программировании и использовалась в проекте языка GJ (Bracha, Odersky, Stoutamire, and Wadler, 1998). Однако теория этого исчисления несколько сложнее, чем в обычной $F_{<}$. (Ghelli, 1997; Baldan, Ghelli, and Raffaetà, 1999) и становится по-настоящему интересной только при включении в систему рекурсивных типов (ни один нерекурсивный тип X не может удовлетворять условию $X <: \{a : \text{Nat}, b : X\}$).

Контексты еще более общего вида, такие как Γ_4 , в котором допускается взаимная рекурсия между типовыми переменными через их верхние границы, тоже существуют. В таких исчислениях каждому новому связыванию переменной, как правило, разрешается вводить произвольное множество неравенств, налагающих ограничения как на новую переменную, так и на уже существующие.

В этой книге мы больше не будем рассматривать F -ограниченную квантификацию, и будем считать, что Γ_2 , Γ_3 и Γ_4 некорректны. Выражаясь формальнее, мы будем требовать, чтобы каждый раз, когда в каком-либо контексте упоминается тип T , все свободные переменные T были связаны в отрезке контекста слева от места, где встречается T .

Подтипы

Каждая типовая переменная в F_{\leq} связана с некоторым ограничением (точно так же, как каждая обыкновенная термовая переменная связана с некоторым типом), и во время проверки подтиповирования и типов необходимо отслеживать эти ограничения. Для этого мы изменяем связывания типов в контекстах и сохраняем верхнюю границу для каждой типовой переменной. Во время проверки подтиповирования эти границы используются для обоснования шагов вида «типовая переменная X является подтиповом типа T , поскольку мы так предположили».

$$\frac{X \leq : T \in \Gamma}{\Gamma \vdash X \leq : T} \quad (\text{S-TVAR})$$

После добавления этого правила получается, что подтиповирование стало *трехместным* отношением — теперь каждое утверждение о подтиповании будет иметь вид $\Gamma \vdash S \leq : T$ (читается как « S является подтиповом T , предполагая Γ »). Такое уточнение также требует добавления контекстов во все остальные правила подтиповирования (см. рис. 26.1).

Помимо нового правила для переменных, требуется также добавить правило подтиповирования для сравнения универсальных типов (S-ALL). На рис. 26.1 приведен более простой вариант, называемый *ядерным правилом* (kernel rule). Согласно этому правилу, границы двух сравниваемых кванторов должны быть одинаковы.

$$\frac{\Gamma, X \leq : U_1 \vdash S_2 \leq : T_2}{\Gamma \vdash \forall X \leq : U_1 . S_2 \leq : \forall X \leq : U_1 . T_2} \quad (\text{S-ALL})$$

Термин «ядерное» восходит к исходной статье Карделли и Вегнера (Cardelli and Wegner, 1985), где этот вариант F_{\leq} назывался «*ядерной Fun*» (Kernel Fun).

Упражнение 26.2.1 [★ →]: Нарисуйте дерево вывода подтиповирования, показывающее, что $B \rightarrow Y \leq : X \rightarrow B$ в контексте $\Gamma = B \leq : Top, X \leq : B, Y \leq : X$.

$\rightarrow \forall <: \text{Top}$ полная

Расширяет $F_{<:}$ (26.1)

Новые правила подтипования		
$\Gamma \vdash S <: T$		
$\Gamma \vdash T_1 <: S_1$	$\Gamma, X <: T_1 \vdash S_2 <: T_2$	
	$\Gamma \vdash \forall X <: S_1 . S_2 <: \forall X <: T_1 . T_2$	(S-ALL)

Рис. 26.2. «Полная» ограниченная квантификация

Типизация

Требуется также уточнить правила типизации для обычных универсальных типов. Эти расширения не представляют труда: в правиле введения для ограниченной квантификации мы переносим границу из абстракции типа в контекст на время проверки типов в теле,

$$\frac{\Gamma, X <: T \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T . t_2 : \forall X <: T . T_2} \quad (\text{T-TAbs})$$

а в правиле устраниния мы проверяем, что указанный тип на самом деле удовлетворяет граничному условию:

$$\frac{\Gamma \vdash t_1 : \forall X <: T_{11} . T_{12} \quad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1 [T_2] : [X \mapsto T_2] X . T_{12}} \quad (\text{T-TAPP})$$

Полная $F_{<:}$

В ядерной $F_{<:}$ два типа с кванторами можно сравнивать только в том случае, когда у них одинаковые верхние границы. Если мы рассматриваем квантор как своего рода функциональный тип (элементами которого являются функции из типов в термы), то ядерное правило соответствует «ковариантному» ограничению стандартного правила подтипования для функциональных типов, в котором область определения исходного типа не может изменяться в подтипах:

$$\frac{S_2 <: T_2}{U \rightarrow S_2 <: U \rightarrow T_2}$$

Такое ограничение выглядит довольно неестественным, как для обычных функций, так и для кванторов. Аналогия подсказывает, что следует уточнить ядерное правило S-ALL и разрешить контравариантное подтиповование «слева от» ограниченных кванторов, как показано на рис. 26.2.

Интуитивно полную версию правила S-ALL можно понимать так: тип $T = \forall X <: T_1 . T_2$ описывает совокупность функций, переводящих типы в значения, каждая из которых отображает подтипы T_1 в экземпляры T_2 . Если T_1

— подтип S_1 , то область определения T меньше, чем у $S = \forall X <: S_1 . S_2$, так что S оказывается более сильным ограничением и описывает меньшую совокупность полиморфных значений. Более того, если для каждого типа U , подходящего в качестве аргументов функциям из обеих совокупностей (т. е. удовлетворяющего более жесткому ограничению $U <: T_1$), U -конкретизация S_2 является подтипов U -конкретизации T_2 , то ограничение S «поточечно сильнее», и, опять же, описывает меньшую совокупность значений.

Система, в которой принято только ядерное правило подтиповирования для кванторных типов, называется «ядерной $F_{<:}$ ». Та же система с полным правилом подтиповирования для кванторов называется «полной $F_{<:}$ ». Простое имя $F_{<:}$ неоднозначно и может относиться к любой из этих двух систем.

Упражнение 26.2.2 [★ →]: Приведите несколько примеров пар типов, которые в полной $F_{<:}$ связаны отношением подтиповирования, а в ядерной $F_{<:}$ — нет.

Упражнение 26.2.3 [★★★★]: Можно ли найти какие-либо полезные примеры, обладающие таким свойством?

26.3. Примеры

В этом разделе дается несколько небольших примеров программирования на $F_{<:}$. В этих примерах мы пытаемся продемонстрировать свойства системы, а не ее практическое применение; более крупные расширенные примеры можно найти в последующих главах (27 и 32). Все примеры этой главы будут работать как в ядерной, так и в полной $F_{<:}$.

Кодирование типов-произведений

В §23.4 мы показали, как закодировать пары чисел в Системе F. Это кодирование легко обобщается на пары произвольных типов: элементы типа

```
Pair T1 T2 = ∀X. (T1 → T2 → X) → X;
```

представляют собой пары из T_1 и T_2 . Конструктор `pair` и деструкторы `fst` и `snd` определяются, как показано ниже. (Приписывание типа в определении `pair` помогает интерпретатору распечатать тип в понятном человеку виде.)

```
pair = λX. λY. λx:X. λy:Y. (λR. λp:X → Y → R. p x y) as Pair X Y;

▷ pair : ∀X. ∀Y. X → Y → Pair X Y

fst = λX. λY. λp: Pair X Y. p [X] (λx:X. λy:Y. x);

▷ fst : ∀X. ∀Y. Pair X Y → X

snd = λX. λY. λp: Pair X Y. p [Y] (λx:X. λy:Y. y);

▷ snd : ∀X. ∀Y. Pair X Y → Y
```

Понятно, что такой же способ кодирования можно использовать и в $F_{<:}$, поскольку $F_{<:}$ содержит все конструкции Системы F. Интереснее, однако, то, что это кодирование обладает некоторыми естественными свойствами в части потипирования. В частности, ожидаемое правило подтиповования для пар

$$\frac{\Gamma \vdash S_1 <: T_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash \text{Pair } S_1 \ S_2 <: \text{Pair } T_1 \ T_2}$$

прямо из него следует.

Упражнение 26.3.1 [$\star \rightarrow$]: Покажите это.

Кодирование записей

Интересно заметить, что записи и типы записей — включая законы подтиповования для них, — можно закодировать в чистой $F_{<:}$. Представленный здесь способ кодирования был открыт Карделли (Cardelli, 1992).

Для начала определим *гибкие кортежи* (flexible tuples). Они называются «гибкими», поскольку при образовании подтипов разрешается расширять их справа, в отличие от обычновенных кортежей.

Определение 26.3.2 Для каждого $n \geq 0$ и типов с T_1 по T_n , пусть

$$\{T_i^{i \in 1..n}\} \stackrel{\text{def}}{=} \text{Pair } T_1 \ (\text{Pair } T_2 \ \dots \ (\text{Pair } T_n \ \text{Top}) \dots)$$

В частности: $\{\}$ $\stackrel{\text{def}}{=} \text{Top}$. Аналогично, для термов с t_1 по t_n , пусть

$$\{t_i^{i \in 1..n}\} \stackrel{\text{def}}{=} \text{pair } t_1 \ (\text{pair } t_2 \ \dots \ (\text{pair } t_n \ \text{top}) \dots)$$

(ради краткости мы опускаем аргументы-типы конструктора *pair*. В качестве терма *top* используется какой-нибудь произвольный элемент *Top* — т. е. произвольный замкнутый правило типизированный терм.) Проекция $t.n$ (опять же, опуская типовые аргументы) равна

$$fst \ (\underbrace{\underbrace{\underbrace{\underbrace{\dots \ (snd \ (\underbrace{\underbrace{\dots \ (snd \ t) \dots})}_{n-1 \ раз})}_{\dots}}}_{\dots})$$

Из такого сокращения мы немедленно получаем следующие правила для подтиповования и типизации гибких кортежей:

$$\frac{\Gamma \vdash S_i^{i \in 1..n} <: T_i}{\Gamma \vdash \{S_i^{i \in 1..n+k}\} <: \{T_i^{i \in 1..n}\}}$$

$$\frac{\Gamma \vdash t_i^{i \in 1..n} : T_i}{\Gamma \vdash \{t_i^{i \in 1..n}\} : \{T_i^{i \in 1..n}\}}$$

$$\frac{\Gamma \vdash t : \{T_i^{i \in 1..n}\}}{\Gamma \vdash t.i : T_i}$$

Пусть теперь имеется счетное множество меток \mathcal{L} , и для него задан полный порядок при помощи биективной функции *метка-с-индексом* : $\mathbb{N} \rightarrow \mathcal{L}$. Определим записи следующим образом.

Определение 26.3.3 Пусть имеется L — конечное подмножество \mathcal{L} , и для каждого $l \in L$ задан тип S_l . Пусть максимальный индекс элементов L равен m , и при этом

$$\hat{S}_i = \begin{cases} S_l & \text{если метка-с-индексом}(i) = l \in L \\ \text{Top} & \text{если метка-с-индексом}(i) \notin L. \end{cases}$$

Тип записей $\{l : S_l^{l \in L}\}$ определяется как гибкий кортеж $\{\hat{S}_i^{i \in 1..m}\}$. Аналогично, если для каждого $l \in L$ имеется терм t_l , то

$$\hat{t}_i = \begin{cases} t_l & \text{если метка-с-индексом}(i) = l \in L \\ \text{top} & \text{если метка-с-индексом}(i) \notin L. \end{cases}$$

Значение записи $\{l = t_l^{l \in L}\}$ определяется как $\{\hat{t}_i^{i \in 1..m}\}$. Проекция $t.l$ — просто проекция кортежа $t.i$, где метка-с-индексом(i) = l .

При таком кодировании выполняются все необходимые правила типизации и подтипования для записей (S-RCDWIDTH, S-RCDDEPTH, S-RCDPERM, T-RCD и T-PROJ по рис. 15.2 и 15.3). Однако интерес к подобному построению в основном теоретический — с практической точки зрения, опора на глобальное упорядочение имен полей является серьезным недостатком, поскольку получается, что в языке с раздельной компиляцией нельзя нумеровать поля отдельно в каждом модуле, а нужно это сделать один раз для всех полей, т. е. при компоновке.

Кодирование по Чёрчу с подтипованием

В качестве последней демонстрации выразительной силы $F_<$, давайте рассмотрим, что получается при добавлении ограниченной квантификации к кодированию чисел по Чёрчу в Системе F (§23.4). Там тип чисел Чёрча был

$$\text{CNat} = \forall X. (X \rightarrow X) \rightarrow X \rightarrow X;$$

Неформально можно прочитать этот тип так: «Скажите мне тип результата T ; потом дайте мне функцию на T и “базовый элемент” T , и тогда я верну вам еще один элемент T , полученный n -кратным применением данной мне функции к данному мне исходному элементу».

Можно обобщить это определение, добавив два ограниченных квантора и уточнив тип параметров s и z .

$$\text{SNat} = \forall X <: \text{Top}. \ \forall S <: X. \ \forall Z <: X. (X \rightarrow S) \rightarrow Z \rightarrow X;$$

Этот тип можно интуитивно прочитать так: «Дайте мне тип обобщенного результата X и два подтипа S и Z . Потом дайте мне функцию, которая отображает все множество X на подмножество S , а также элемент особого множества Z , и я верну вам элемент X , получаемый n -кратным применением функции к базовому элементу».

Чтобы понять, чем интересно это уточнение, рассмотрим следующий, несколько отличающийся тип:

```
SZero = ∀X<:Top. ∀S<:X. ∀Z<:X. (X → S) → Z → Z;
```

Несмотря на то, что `SZero` имеет почти такой же вид, как `SNat`, он говорит о поведении своих элементов намного больше, поскольку обещает, что окончательный результат будет элементом `Z`, а не просто `X`. В сущности, есть только один способ обеспечить такое поведение — а именно, вернуть сам аргумент `z`. Другими словами, значение

```
szero = λX. λS<:X. λZ<:X. λs:X → S. λz:Z. z;
```

```
> szero : SZero
```

является *единственным* элементом типа `SZero` (в том смысле, что всякий другой элемент типа `SZero` по поведению неотличим от `szero`). Поскольку `SZero` — подтип `SNat`, имеем также `szero : SNat`.

С другой стороны, в похожем типе

```
SPos = ∀X<:Top. ∀S<:X. ∀Z<:X. (X → S) → Z → S;
```

содержится больше различных элементов, например,

```
sone = λX. λS<:X. λZ<:X. λs:X → S. λz:Z. s z;
stwo = λX. λS<:X. λZ<:X. λs:X → S. λz:Z. s (s z);
sthree = λX. λS<:X. λZ<:X. λs:X → S. λz:Z. s (s (s z));
```

и т.д. Действительно, `SPos` содержит все элементы `SNat` за исключением `szero`.

Аналогично, можно уточнить типы операций над числами Чёрча. Например, с помощью системы типов можно проверить, что результатом функции-последователя всегда является положительное число.

```
ssucc = λn:SNat.
        λX. λS<:X. λZ<:X. λs:X → S. λz:Z.
        s (n [X] [S] [Z] s z);
```

```
> ssucc : SNat → SPos
```

Точно так же, уточняя типы параметров, мы можем написать функцию `plus` так, чтобы процедура проверки типов приписывала ей тип `SPos → SPos → SPos`.

```
pluspp = λn:SPos. λm:SPos.
          λX. λS<:X. λZ<:X. λs:X → S. λz:Z.
          n [X] [S] [Z] s (m [X] [S] [Z] s z);
```

```
> pluspp : SPos → SPos → SPos
```

Упражнение 26.3.4 [★★]: Напишите еще один вариант функции `plus`, отличающийся от приведенного выше только аннотациями типов, и имеющий тип `SZero → SZero → SZero`. Напишите вариант с типом `SPos → SNat → SPos`.

Последний пример и упражнение к нему обращают внимание на интересный вопрос. Ясно, что мы не хотим писать несколько различных версий функции `plus` с различными именами, а потом решать, которую из них применить в соответствии с типами аргументов: хотелось бы иметь *одну* версию `plus`, в типе которой учитывались бы все эти возможности — нечто вроде

```

plus : SZero → SZero → SZero
      ∧ SNat→SPos→SPos
      ∧ SPos→SNat→SPos
      ∧ SNat→SNat→SNat

```

где запись $t : S \wedge T$ означает «терм t имеет одновременно типы S и T ». Желание поддержать в языке такую перегрузку типов привело к исследованию систем, сочетающих *типы-пересечения* (intersection types) (15.7) с ограниченной квантификацией; см. Pierce, 1997b.

Упражнение 26.3.5 [РЕКОМЕНДУЕТСЯ, ★★]: Действуя по образцу $SNat$ и компании, обобщите тип чёрчевых булевских значений $CBool$ (§23.4) до типа $SBool$ с двумя подтипами $STrue$ и $SFalse$. Напишите функцию $notft$ с типом $SFalse \rightarrow STrue$ и функцию $nottf$ с типом $STrue \rightarrow SFalse$.

Упражнение 26.3.6 [★ →]: В начале главы мы упомянули, что подтипы и полиморфизм можно совместить более простым и ортогональным способом, чем в $F_{<:}$. Начнем с Системы F (возможно, с добавлением записей и т. п.) и добавим отношение подтипов (как в простом типизированном лямбда-исчислении с подтипами), но оставим квантификацию неограниченной. Единственным расширением отношения подтипов будет ковариантное правило подтипов для тел обычных кванторов:

$$\frac{S <: T}{\forall X. S <: \forall X. T}$$

Какие примеры из данной главы можно сформулировать в рамках этой более простой системы?

26.4. Безопасность

Свойство сохранения типов можно доказать обычным способом как для ядерного, так и для полного варианта $F_{<:}$. Здесь мы приводим подробное доказательство для ядерной $F_{<:}$; для полной $F_{<:}$ рассуждение выглядит почти так же. Однако когда в главе 28 мы рассмотрим алгоритмы проверки типов и подтипов, окажется, что два варианта различаются сильнее, чем можно предположить по простым доказательствам из этой главы. Мы увидим, что во многих деталях полная система намного труднее поддается анализу, чем ядерная, и даже что полная система не обладает некоторыми полезными свойствами (включая разрешимость проверки типов!), которые в ядерной системе присутствуют.

Вначале мы устанавливаем некоторые предварительные технические леммы, касающиеся отношений типизации и подтипов. Все они доказываются обычной индукцией по деревьям вывода.

Лемма 26.4.1 [ПЕРЕСТАНОВКА]: Предположим, что Γ — правильно сформированный контекст, а Δ — перестановка Γ , то есть Δ связывает те же самые переменные, что и Γ , и порядок этих связываний в Δ сохраняет области видимости типовых переменных из Γ в том смысле, что если одно

связывание в Γ вводит типовую переменную, которая упоминается в другом связывании, расположеннном правее, то в Δ порядок этих связываний сохраняется.

1. Если $\Gamma \vdash t : T$, то $\Delta \vdash t : T$.
2. Если $\Gamma \vdash S <: T$, то $\Delta \vdash S <: T$.

Лемма 26.4.2 [ОСЛАБЛЕНИЕ]:

1. Если $\Gamma \vdash t : T$ и контекст $\Gamma, x:U$ сформирован правильно, то $\Gamma, x:U \vdash t : T$.
2. Если $\Gamma \vdash t : T$ и контекст $\Gamma, X <: U$ сформирован правильно, то $\Gamma, X <: U \vdash t : T$.
3. Если $\Gamma \vdash S <: T$ и контекст $\Gamma, x:U$ сформирован правильно, то $\Gamma, x:U \vdash S <: T$.
4. Если $\Gamma \vdash S <: T$ и контекст $\Gamma, X <: U$ сформирован правильно, то $\Gamma, X <: U \vdash S <: T$.

Упражнение 26.4.3 [★]: В каком месте доказательство свойства ослабления основано на перестановке?

Лемма 26.4.4 [УСИЛЕНИЕ ДЛЯ ТЕРМОВЫХ ПЕРЕМЕННЫХ В ДЕРЕВЬЯХ ВЫВОДА ПОДТИПОВ]: Если $\Gamma, x:T, \Delta \vdash S <: T$, то $\Gamma, \Delta \vdash S <: T$.

Доказательство: Утверждение очевидно: предположения о типизации при выводе подтипов не играют никакой роли.

Как обычно, доказательство сохранения типов опирается на несколько лемм, связывающих подстановку с отношениями типизации и подтипования.

Лемма 26.4.5 [Сужение]:

1. Если $\Gamma, X <: Q, \Delta \vdash S <: T$ и $\Gamma \vdash P <: Q$, то $\Gamma, X <: P, \Delta \vdash S <: T$.
2. Если $\Gamma, X <: Q, \Delta \vdash t : T$ и $\Gamma \vdash P <: Q$, то $\Gamma, X <: P, \Delta \vdash t : T$.

Эти леммы часто называют свойствами сужения (*narrowing*), поскольку они связаны с ограничением (сужением) возможных значений переменной X . **Доказательство:** УПРАЖНЕНИЕ [★].

Затем, как обычно, требуется лемма, связывающая подстановку и отношение типизации.

Лемма 26.4.6 [ПОДСТАНОВКА СОХРАНЯЕТ ТИПИЗАЦИЮ]: Если $\Gamma, x:Q, \Delta \vdash t : T$ и $\Gamma \vdash q : Q$, то $\Gamma, \Delta \vdash [x \mapsto q]t : T$.

Доказательство: Индукция по дереву вывода $\Gamma, x:Q, \Delta \vdash t : T$ с использованием свойств, доказанных ранее.

Поскольку при вычислении мы будем иногда подставлять типы вместо типовых переменных, нам требуется также лемма, связывающая подстановку типов с типизацией. Ее доказательство (а именно, вариант T-SUB) зависит от новой леммы, связывающей подстановку и подтиповование.

Определение 26.4.7 Запись $[X \mapsto S]\Gamma$ обозначает контекст, получающий подстановкой S вместо X в правых частях всех связываний в Γ .

Лемма 26.4.8 [Подстановка типов сохраняет отношение подтиповования]:
Если $\Gamma, X<:Q, \Delta \vdash S <: T$ и $\Gamma \vdash P <: Q$, то $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$.

Обратите внимание, что требуется проводить подстановку для X только в той части окружения, которая следует за связыванием X , поскольку наши соглашения об областях видимости требуют, чтобы типы слева от связывания X не содержали переменной X .

Доказательство: Индукция по дереву вывода $\Gamma, X<:Q, \Delta \vdash S <: T$. Интерес представляют только два последних варианта:

Вариант S-TVAR: $S = Y \quad Y <: T \in (\Gamma, X<:Q, \Delta)$

Нужно рассмотреть два подварианта. Если $Y \neq X$, то результат немедленно следует из S-TVAR. Если же $Y = X$, то имеем $T = Q$ и $[X \mapsto P]S = Q$, и тогда результат следует из S-REFL.

Вариант S-ALL: $S = \forall Z <: U_1, S_2 \quad T = \forall Z <: U_1 . T_2$

$\Gamma, X <: Q, \Delta, Z <: U_1 \vdash S_2 <: T_2$

Согласно предположению индукции, $\Gamma, [X \mapsto P]\Delta, Z <: [X \mapsto P]U_1 \vdash [X \mapsto P]S_2 <: [X \mapsto P]T_2$. По правилу S-ALL, $\Gamma, [X \mapsto P]\Delta \vdash \forall Z <: [X \mapsto P]U_1 . [X \mapsto P]S_2 <: \forall Z <: [X \mapsto P]U_1 . [X \mapsto P]T_2$, то есть, как и требуется, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P](\forall Z <: U_1 . S_2) <: [X \mapsto P](\forall Z <: U_1 . T_2)$.

Аналогичная лемма связывает подстановку типов и типизацию.

Лемма 26.4.9 [Подстановка типов сохраняет типизацию]:
Если $\Gamma, X <: Q, \Delta \vdash t : T$ и $\Gamma \vdash P <: Q$, то $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t : [X \mapsto P]T$.

Доказательство: Индукция по дереву вывода $\Gamma, X <: Q, \Delta \vdash t : T$. Мы приводим только интересные варианты.

Вариант T-TAPP: $t = t_1 [T_2] \quad \Gamma, X <: Q, \Delta \vdash t_1 : \forall Z <: T_{11} . T_{12}$

$T = [Z \mapsto T_2]T_{12}$

По предположению индукции,

$$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t_1 : [X \mapsto P](\forall Z <: T_{11} . T_{12})$$

m. e.

$$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t_1 : \forall Z <: T_{11} . [X \mapsto P]T_{12}$$

По правилу T-TAPP,

$$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t_1 [X \mapsto P]T_2 : [Z \mapsto [X \mapsto P]T_2]([X \mapsto P]T_{12})$$

m. e.

$$\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P](t_1 [T_2]) : [X \mapsto P]([Z \mapsto T_2]T_{12})$$

Вариант T-SUB: $\Gamma, X<:q, \Delta \vdash t : S\Gamma, X<:q, \Delta \vdash S <: T$

По предположению индукции, $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t : [X \mapsto P]T$. По лемме о сохранении отношения подтипования при подстановке (лемма 26.4.8) имеем $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$, и требуемый результат следует по правилу T-SUB.

Докажем теперь некоторые простые структурные свойства подтипования.

Лемма 26.4.10 [Инверсия отношения подтипования, справа налево]:

1. Если $\Gamma \vdash S <: X$, то S является типовой переменной.
2. Если $\Gamma \vdash S <: T_1 \rightarrow T_2$, то либо S — типовая переменная, либо S имеет вид $S_1 \rightarrow S_2$, причем $\Gamma \vdash T_1 <: S_1$ и $\Gamma \vdash S_2 <: T_2$.
3. Если $\Gamma \vdash S <: \forall X <: U_1 . T_2$, то либо S — типовая переменная, либо S имеет вид $\forall X <: U_1 . S_2$, причем $\Gamma, X <: U_1 \vdash S_2 <: T_2$.

Доказательство: Пункт (1) доказывается простой индукцией по деревьям вывода подтипования. Единственный интересный вариант — правило S-TRANS, где предположение индукции надо применить два раза, сначала к правой предпосылке, а затем к левой. Доказательства остальных пунктов строятся так же, с использованием пункта (1) в варианте с правилом транзитивности.

Упражнение 26.4.11 [РЕКОМЕНДУЕТСЯ, ★★]: Докажите следующие «свойства инверсии слева направо»:

1. Если $\Gamma \vdash S_1 \rightarrow S_2 <: T$, то либо $T = \text{Top}$, либо $T = T_1 \rightarrow T_2$, причем $\Gamma \vdash T_1 <: S_1$ и $\Gamma \vdash S_2 <: T_2$.
2. Если $\Gamma \vdash \forall X <: U . S_2 <: T$, то либо $T = \text{Top}$, либо $T = \forall X <: U . T_2$, причем $\Gamma, X <: U \vdash S_2 <: T_2$.
3. Если $\Gamma \vdash X <: T$, то либо $T = \text{Top}$, либо $T = X$, либо $\Gamma \vdash S <: T$ при $X <: S \in \Gamma$.
4. Если $\Gamma \vdash \text{Top} <: T$, то $T = \text{Top}$.

Лемма 26.4.10, в свою очередь, используется при доказательстве следующего несложного структурного свойства отношения типизации, которое требуется в ключевых моментах при доказательстве свойства сохранения.

Лемма 26.4.12

1. Если $\Gamma \vdash \lambda x:S_1 . s_2 : T$ и $\Gamma \vdash T <: U_1 \rightarrow U_2$, то $\Gamma \vdash U_1 <: S_1$ и существует некоторый тип S_2 , такой, что $\Gamma, x:S_1 \vdash s_2 : S_2$ и $\Gamma \vdash S_2 <: U_2$.

2. Если $\Gamma \vdash \lambda X <: S_1 . s_2 : T$ и $\Gamma \vdash T <: \forall X <: U_1 . U_2$, то $U_1 = S_1$ и существует некоторый тип S_2 , такой, что $\Gamma, X <: S_1 \vdash s_2 : S_2$ и $\Gamma, X <: S_1 \vdash S_2 <: U_2$.

Доказательство: Прямолинейная индукция по деревьям вывода типов, с использованием леммы 26.4.10 при шаге индукции (вариант с правилом T-SUB).

Имея в своем распоряжении все эти утверждения, нетрудно доказать теорему о сохранении типов.

Теорема 26.4.13 [СОХРАНЕНИЕ]: Если $\Gamma \vdash t : T$ и $t \rightarrow t'$, то $\Gamma \vdash t' : T$.

Доказательство: Индукция по дереву вывода $\Gamma \vdash t : T$. При наличии вспомогательных лемм все варианты не представляют особой сложности.

Варианты T-VAR, T-Abs, T-TABS: $t = x$ $t = \lambda x : T_1 . t_2$ или $t = \lambda X <: U . t$
 Эти варианты вообще не могут возникнуть, поскольку мы предположили, что $t \rightarrow t'$, а правил вычисления для переменных, абстракций и абстракций типа не существует.

Вариант T-APP: $t = t_1 \ t_2$ $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$ $T = T_{12}$ $\Gamma \vdash t_2 : T_{11}$
 Исходя из определения отношения вычисления, имеются три подварианта:

Подвариант: $t_1 \rightarrow t'_1$ $t' = t'_1 \ t_2$

Результат следует из предположения индукции и правила T-APP.

Подвариант: t_1 является значением $t_2 \rightarrow t'_2$ $t' = t_1 \ t'_2$

Аналогично.

Подвариант: $t_1 = \lambda x : U_{11} . u_{12}$ $t' = [x \mapsto t_2]u_{12}$

Согласно лемме 26.4.12, $\Gamma, x : U_{11} \vdash u_{12} : U_{12}$ для некоторого U_{12} , такого, что $\Gamma \vdash T_{11} <: U_{11}$ и $\Gamma \vdash U_{12} <: T_{12}$. По лемме о сохранении типизации при подстановке (лемма 26.4.6), $\Gamma \vdash [x \mapsto t_2]u_{12} : U_{12}$, откуда получаем $\Gamma \vdash [x \mapsto t_2]u_{12} : T_{12}$ по правилу T-SUB.

Вариант T-TAPP: $t = t_1 [T_2]$ $\Gamma \vdash t : \forall X <: T_{11} . T_{12}$

$T = [X \mapsto T_2]T_{12}$ $\Gamma \vdash T_2 <: T_{11}$

Исходя из определения отношения вычисления, имеется два подварианта:

Подвариант: $t_1 \rightarrow t'_1$ $t' = t'_1 [T_2]$

Результат следует из предположения индукции и правила T-TAPP.

Подвариант: $t_1 = \lambda X <: U_{11} . u_{12}$ $t' = [X \mapsto T_2]u_{12}$

Согласно лемме 26.4.12, $U_{11} = T_{11}$ и $\Gamma, X <: U_{11} \vdash u_{12} : U_{12}$, причем $\Gamma, X <: U_{11} \vdash U_{12} <: T_{12}$. По свойству сохранения типизации при подстановке (лемма 26.4.9), $\Gamma \vdash [X \mapsto T_2]u_{12} : [X \mapsto T_2]U_{12}$, откуда по лемме 26.4.8 и правилу T-SUB имеем $\Gamma \vdash [X \mapsto T_2]u_{12} : [X \mapsto T_2]T_{12}$.

Вариант Т-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

По предположению индукции, $\Gamma \vdash t' : S$, и требуемый результат получается по правилу Т-SUB.

Теорема о продвижении для $F_<$, является прямолинейным расширением такой же теоремы для простого типизированного лямбда-исчисления с подтипами. Как всегда, вначале мы выписываем лемму о канонических формах, которая говорит, какой вид могут иметь замкнутые значения функциональных и квантифицированных типов.

Лемма 26.4.14 [Канонические формы]:

1. Если v — замкнутое значение типа $T_1 \rightarrow T_2$, то v имеет вид $\lambda x:S_1. t_2$.
2. Если v — замкнутое значение типа $\forall X<:T_1. T_2$, то v имеет вид $\lambda X<:T_1. t_2$.

Доказательство: В обоих пунктах проводится индукция по деревьям вывода типов; мы приводим доказательство только для второй части. При рассмотрении правил типизации становится ясно, что последнее правило в выводе $\vdash v : \forall X<:T_1. T_2$ может быть только Т-TABS или Т-SUB. Если это Т-TABS, то требуемый результат немедленно следует из предпосылки правила. Предположим, что последнее правило Т-SUB. Из предпосылок этого правила мы имеем $\vdash v : S$ и $S <: \forall X<:T_1. T_2$. Из леммы об инверсии (26.4.10) мы знаем, что S имеет вид $\forall X<:T_1. S_2$. Теперь результат следует из предположения индукции.

При наличии этой леммы доказательство свойства продвижения не представляет труда.

Теорема 26.4.15 [ПРОДВИЖЕНИЕ] Если имеется замкнутый правильно типизированный $F_<$ -терм t , то либо t является значением, либо существует некоторый терм t' такой, что $t \rightarrow t'$.

Доказательство: Индукция по деревьям вывода типов. Вариант с переменной возникнуть не может, поскольку терм t замкнут. Два варианта с лямбда-абстракциями следуют непосредственно, поскольку абстракции термов и типов являются значениями. Варианты с применением, применением типов и включением более интересны; мы демонстрируем только два последних (применение термов подобно применению типов).

*Вариант Т-TAPP: $t = t_1 [T_2] \quad \vdash t_1 : \forall X<:T_{11}. T_{12}$
 $\Gamma \vdash T_2 <: T_{11} \quad T = [X \mapsto T_2] T_{12}$*

Согласно предположению индукции, либо терм t_1 является значением, либо он способен произвести шаг вычисления. Если он может сделать шаг, то к t применимо правило Т-TAPP. В противном случае, если t_1 — значение, то из пункта (2) леммы о канонических формах (26.4.14) нам известно, что t_1 имеет вид $\lambda X<:T_{11}. t_{12}$, так что к t оказывается применимо правило Е-TAPP TABS.

Вариант Т-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

Результат следует напрямую из предположения индукции.

→ ∀ <: Тор \exists Расширяет $F_{<}$ (26.1) и неограниченную экзистенциальную квантификацию (24.1)

Новые синтаксические формы

$T ::= \dots$ типы:
 $\{\exists X <: T, t\}$ экзистенциаль-
 ный тип

Новые правила подтипования

$\Gamma \vdash S <: T$

Новые правила типизации $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2 \quad \Gamma \vdash U <: T_1}{\Gamma \vdash \{*U, t_2\} \text{ as } \{\exists X <: T_1, T_2\} : \{\exists X <: T_1, T_2\}} \quad (\text{T-PACK})$$

$$\frac{\Gamma, X <: U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X <: U, S_2\} <: \{\exists X <: U, T_2\}} \quad (\text{S-SOME})$$

$$\frac{\Gamma \vdash t_1 : \{\exists X <: T_{11}, T_{12}\} \quad \Gamma, X <: T_{11}, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2} \quad (\text{T-UNPACK})$$

Рис. 26.3. Ограничные кванторы существования (ядерный вариант)

Упражнение 26.4.16 [★★★ →]: Адаптируйте доказательство из этого раздела к полной $F_{<}$.

26.5. Ограничные экзистенциальные типы

К экзистенциальным типам (глава 24) можно добавить ограничения точно так же, как мы это сделали для универсальных типов. При этом получаются *ограниченные экзистенциальные типы* (bounded existentials), изображенные на рис. 26.3. Как и в случае с ограниченными универсальными типами, правило подтипования S-SOME имеет две разновидности — в одной границы сравниваемых кванторов обязаны быть одинаковыми, во второй они могут различаться.

Упражнение 26.5.1 [★]: Как выглядит полный вариант правила S-SOME?

Упражнение 26.5.2 [★]: Сколько можно найти в чистой Системе F с записями и экзистенциальными типами (но без подтипов) таких способов выбора T , чтобы

$\{*Nat, \{a=5, b=7\}\} \text{ as } T;$

был правильно типизирован? Появляются ли новые варианты при добавлении подтипов и ограниченных экзистенциальных типов?

В §24.2 мы видели, как с помощью обычных экзистенциальных типов можно реализовать абстрактные типы данных. Когда мы добавляем ограничения к кванторам существования, этот прирост выразительности отражается

на уровне АТД. Карделли и Вегнер (Cardelli and Wegner, 1985) назвали такие конструкции *частично абстрактными типами* (partially abstract types). Интуитивная идея здесь состоит в том, что ограниченный квантор существования сообщает внешнему миру *часть* информации о типе представления, но точное устройство типа представления остается неизвестным.

Например, предположим, что мы реализовали АТД счетчиков, как в §24.2, но добавили в аннотацию типа ограничение `Counter<:Nat`.

```
counterADT =
  {*Nat, {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i)}}
  as {∃Counter<:Nat,
      {new: Counter, get: Counter → Nat, inc: Counter →
      Counter}};

▷ counterADT : {∃Counter<:Nat,
                 {new: Counter, get: Counter →
                 Nat, inc: Counter → Counter}}
```

АТД счетчиков может использоваться в точности как раньше, путем связывания его типовой и термовой компонент переменными `Counter` и `counter` и последующего обращения к полям `counter` для доступа к операциям над счетчиками

```
let {Counter,counter} = counterADT in
  counter.get (counter.inc (counter.inc counter.new));

▷ 3 : Nat
```

Более того, теперь нам разрешается использовать значения `Counter` напрямую как числа:

```
let {Counter,counter} = counterADT in
  succ (succ (counter.inc counter.new));

▷ 4 : Nat
```

С другой стороны, мы по-прежнему не имеем права использовать числа как значения типа `Counter`:

```
let {Counter,counter} = counterADT in
  counter.inc 3;

▷ Ошибка: несоответствие типа параметра
```

В сущности, в этой версии абстракции счетчика мы решили облегчить внешнему миру использование счетчиков, открыв внутреннее представление типа; однако мы сохраняем контроль над тем, как счетчики создаются.

Упражнение 26.5.3 [★ ★ ★]: Предположим, мы хотим определить два абстрактных типа данных, `Counter` и `ResetCounter`, так что а) оба АТД имеют операции `new`, `get` и `inc`; б) тип `ResetCounter` дополнительно предоставляет операцию `reset`, принимающую счетчик и возвращающую новый счетчик, установленный в какое-то заранее фиксированное состояние, например, 1; в) клиентам обоих АТД разрешено использовать `ResetCounter`

вместо *Counter* (т. е. выполняется *ResetCounter <: Counter*); и 2) клиентам не дается больше никакой информации о том, как счетчики и счетчики со сбросом представлены внутри. Можно ли выполнить все эти условия при помощи ограниченных экзистенциальных пакетов?

Можно аналогично адаптировать и наш способ кодирования объектов на основе экзистенциальных типов из §24.2. При таком кодировании типы-свидетели экзистенциальных пакетов использовались для представления типов внутреннего состояния объектов, и это состояние всегда было записью, состоящей из переменных экземпляра. Заменив неограниченный экзистенциальный тип ограниченным, мы можем показать внешнему миру имена и типы некоторых, но не всех, переменных экземпляра объекта. Например, вот объект-счетчик с частично видимым внутренним состоянием, показывающий только поле *x*, но ограничивающий доступ к (не слишком интересному) полю *private*:

```
c = {*{x:Nat, private:Bool},
      {state = {x=5, private=false},
       methods = {get = λs:{x:Nat}. s.x,
                  inc = λs:{x:Nat, private:Bool}.
                         {x=succ(s.x), private=s.private}}}}
     as {∃X<:{x:Nat}, {state:X, methods: {get:X → Nat, inc:X →
X}}};
```



```
▷ c : {∃X<:{x:Nat}, {state:X, methods: {get:X → Nat, inc:X →
X}}}
```

Как и в случае с частично абстрактным АТД счетчиков, наш счетчик-объект позволяет обращаться к своему значению как через вызов метода *get*, так и напрямую, путем чтения своего поля *x*.

Упражнение 26.5.4 [★★]: Покажите, как расширить кодирование экзистенциальных типов универсальными из §24.3 на случай кодирования ограниченных экзистенциальных типов через ограниченные универсальные. Убедитесь, что правило подтиповования S-SOME следует из кодирования и правил подтиповирования для ограниченных кванторов.

26.6. Дополнительные замечания

Язык CLU (Liskov et al., 1977, 1981; Schaffert, 1978; Scheifler, 1978) был, по-видимому, первым языком с безопасной ограниченной квантификацией. Понятие границ параметров в CLU по существу представляет собой ограниченную квантификацию (§26.2), обобщенную до множественных параметров-типов.

В форме, рассматриваемой в этой книге, идея ограниченной квантификации была впервые введена Карделли и Вегнером в языке Fun (Cardelli and Wegner, 1985). Их исчисление «ядерной Fun» соответствует нашей ядерной системе *F_{<:}*. Fun был основан на более ранних идеях Карделли и формализован при помощи методов, разработанных Митчеллом (Mitchell, 1984b); он считал

полиморфизм по Жирару-Рейнольдсу (Girard, 1972; Reynolds, 1974) с исчислением подтипов первого порядка Карделли (Cardelli, 1984). Исходная версия Fun была упрощена и подверглась некоторому обобщению в работе Брюса и Лонго (Bruce and Longo, 1990), а затем в работе Куриена и Гелли (Curien and Ghelli, 1992). В результате получилось исчисление, которое мы называем полной $F_{<:}$. Наиболее подробным обзором по ограниченной квантификации является статья Карделли, Мартини, Митчелла и Щедрова (Cardelli, Martini, Mitchell, and Scedrov, 1994).

$F_{<:}$ и родственные системы активно изучались теоретиками и разработчиками языков программирования. Первые примеры ограниченной квантификации были даны в статье-обзоре Карделли и Вегнера; дальнейшие примеры последовали в работе Карделли по степенным видам (power kinds) (Cardelli, 1988a). Куриен и Гелли (Curien and Ghelli, 1992; Ghelli, 1990) рассматривают некоторые синтаксические свойства $F_{<:}$. Семантические свойства систем, близких к $F_{<:}$, изучаются в работах Брюса и Лонго (Bruce and Longo, 1990), Мартини (Martini, 1988), Бреазу-Таннена, Кокана, Гантера и Щедрова (Breazu-Tannen, Coquand, Gunter, and Scedrov, 1991), Кардоне (Cardone, 1989), Карделли и Лонго (Cardelli and Longo, 1991), Карделли, Мартини, Митчелла и Щедрова (Cardelli, Martini, Mitchell, and Scedrov, 1994), Куриена и Гелли (Curien and Ghelli, 1992, 1991), а также Брюса и Митчелла (Bruce and Mitchell, 1992).

Карделли и Митчелл (Cardelli and Mitchell, 1991), Брюс (Bruce, 1991), Карделли (Cardelli, 1992), а также Кэннинг, Кук, Хилл, Олтхофф и Митчелл (Canning, Cook, Hill, Olthoff, and Mitchell, 1989b) расширили $F_{<:}$, добавив типы-записи и более богатое понятие наследования реализации. Ограниченная квантификация играет ключевую роль в языке Quest, созданном Карделли (Cardelli, 1991; Cardelli and Longo, 1991); в языке Abel, разработанном в лабораториях HP (Canning, Cook, Hill, and Olthoff, 1989a; Canning, Cook, Hill, Olthoff, and Mitchell, 1989b; Canning, Hill, and Olthoff, 1988; Cook, Hill, and Canning, 1990); а также в таких недавних разработках, как GJ (Bracha, Odersky, Stoutamire, and Wadler, 1998), Pict (Pierce and Turner, 2000) и Funnel (Odersky, 2000).

Воздействие ограниченной квантификации на кодирование по Чёрчу (§26.3) рассматривалось у Гелли (Ghelli, 1990), а также Карделли, Мартини, Митчеллом и Щедровым (Cardelli, Martini, Mitchell, and Scedrov, 1994).

Расширение $F_{<:}$ типами-пересечениями (15.7) исследовалось Пирсом (Pierce, 1991b, 1997b). Вариант той же системы с видами высшего порядка (higher kinds) применяется к моделированию объектно-ориентированных языков с множественным наследованием реализаций в работе Компаньони и Пирса (Compagnoni and Pierce, 1996); метатеоретические свойства таких языков анализировал Компаньони (Compagnoni, 1994).

Глава 27

Расширенный пример: еще раз императивные объекты

В главе 18 мы разработали набор идиом на языке простого типизированного исчисления с записями, ссылками и подтипами. Эти идиомы моделируют основные черты и приемы императивного объектно-ориентированного стиля программирования. В конце этой главы (§18.12) мы приложили некоторые усилия для того, чтобы повысить эффективность наших объектов. Для этого мы перенесли действия по построению таблицы методов объекта с времени вызова метода на время создания объекта. В этой главе мы с помощью ограниченной квантификации проводим дальнейшие улучшения эффективности нашей модели.*

Примерами в этой главе служат термы F_{\leq} , с записями (рис. 15.3) и ссылками (13.1). Соответствующая реализация на OCaml называется `fullfsubref`.

* В списке ошибок и опечаток на сайте книги (<http://www.cis.upenn.edu/~bcpierce/tapl/errata.txt>) приводятся следующие соображения:

Глава 27 в целом не слишком убедительна. Цель использования F_{\leq} , вроде бы, состоит в том, чтобы создавать таблицу методов один раз для каждого класса, а не для каждого объекта. Код действительно создает нечто по одному разу для каждого класса, однако это нечто является *функцией*, которая создает таблицу методов *при каждом открытом рекурсивном вызове через self!* (См. вызовы `(!self r)`, которые происходят при обращении к методам.) Другими словами, эффективность в итоге оказывается даже хуже, чем та, что мы получили в конце главы 18.

Я благодарен Джону Тангу Бойланду, указавшему на это обстоятельство. Джон также предлагает другой расширенный пример, использующий F_{\leq} в стиле главы 32 для реализации объектов в императивном стиле. Набросок основной конструкции можно найти здесь:

<http://www.cis.upenn.edu/~bcpierce/tapl/boyland-object-encoding.txt>

Код, реализующий эту идею, находится по адресам

<http://www.cs.uwm.edu/classes/cs790/types-f2003/Chapter27-replacement.txt>
<http://www.cs.uwm.edu/classes/cs790/types-f2003/fullfomsubref.tar.gz>

— прим. перев.

Основная идея §18.12 заключалась в том, что классу при вызове метода передавалась ссылка на «таблицу методов `self`». Класс использовал эту ссылку при определении своих методов, а затем мы подправляли ссылку так, чтобы она указывала на заполненную таблицу методов, возвращенную классом. Например, если `SetCounter` и `SetCounterRep` — это публичный интерфейс и тип внутреннего представления для класса объектов-счетчиков с методами `get`, `set` и `inc`,

```
SetCounter = {get:Unit → Nat, set:Nat → Unit, inc:Unit →
Unit};
```

```
CounterRep = {x: Ref Nat};
```

то мы можем реализовать класс счетчиков с присваиванием так:

```
setCounterClass =
λr:CounterRep. λself: Source SetCounter.
{get = λ_:Unit. !(r.x),
set = λi:Nat. r.x:=i,
inc = λ_:Unit. (!self).set
(succ ((!self).get unit))};
```

```
▷ setCounterClass : CounterRep → (Source SetCounter) →
SetCounter
```

Для параметра `self` мы используем тип `Source SetCounter`, а не `Ref SetCounter`, потому что, когда мы определяем подкласс `SetCounterClass`, `self` этого нового класса будет иметь другой тип. Например, если `InstrCounter` и `InstrCounterRep` — типы интерфейса и представления класса объектов-счетчиков с подсчетом доступа

```
instrCounter = {get:Unit → Nat, set:Nat → Unit,
inc:Unit → Unit, accesses:Unit → Nat};
```

```
InstrCounterRep = {x: Ref Nat, a: Ref Nat};
```

то сам класс можно определить так:

```
instrCounterClass =
λr:InstrCounterRep. λself: Source InstrCounter.
let super = setCounterClass r self in
{get = super.get,
set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
inc = super.inc,
accesses = λ_:Unit.
!(r.a)};
```

```
▷ instrCounterClass : InstrCounterRep →
(Source InstrCounter) → InstrCounter
```

Типом параметра `self` здесь будет `Source InstrCounter`, и нам нужно иметь возможность преобразовать `Source InstrCounter` в `Source SetCounter`, чтобы передать этот `self` как аргумент `self` класса `setCounterClass` при построении `super`. Ковариантный конструктор `Source` разрешает такое преобразование, тогда как с инвариантным конструктором `Ref` оно было бы запрещено.

Однако, как мы заметили в конце §18.12, эффективность этой модели классов по-прежнему не оптимальна. Поскольку с каждым объектом, порожденным определенным классом, связаны идентичные таблицы, должно быть можно построить такую таблицу всего один раз, во время создания *класса*, и использовать ее каждый раз при создании объекта. При этом более точно смоделированными окажутся соглашения по реализации настоящих объектно-ориентированных языков: в них объект не несет при себе никаких методов, а только указатель на структуру данных, которая представляет его класс, и именно в этой структуре хранятся методы.¹

То же самое можно выразить еще одним способом, заметив, что порядок параметров в вышеприведенных классах (сначала переменные экземпляра, а потом `self`) неверный: параметр `self` требуется для построения таблицы класса, а запись переменных экземпляра `r` используется только тогда, когда у готового объекта вызываются методы. Если бы `self` был первым аргументом, мы могли бы вычислять таблицу методов до передачи аргумента `r`; можно было бы сначала один раз частично применить класс к его аргументу `self`, проделать это вычисление раз и навсегда и сделать множество копий получившейся таблицы методов, применяя ее ко многим записям переменных экземпляра. Говоря более конкретно, нам хотелось бы переписать `setCounterClass` так:

```
setCounterClass =
  λself: Source (CounterRep → SetCounter).
  λr: CounterRep.
    {get = λ_:Unit. !(r.x),
     set = λi:Nat. r.x:=i,
     inc = λ_:Unit. (!self r).set
      (succ ((!self r).get unit))};

▷ setCounterClass : (Source (CounterRep → SetCounter)) →
  CounterRep → SetCounter
```

Между этой версией и предыдущей есть три существенных отличия. Во-первых, новая версия принимает сначала `self`, а уже потом — `r`. Во-вторых, тип `self` поменялся с `SetCounter` на `CounterRep -> SetCounter`. Это второе изменение вызвано первым, поскольку тип `self` должен быть таким же, как и тип таблицы методов, возвращаемой классом. И в-третьих, все вызовы `!self` в теле класса превращаются в `(!self r)`. Это третье изменение вызвано вторым.

Функция создания объекта для наших счетчиков определена так:

```
newSetCounter =
  let m = ref (λr:CounterRep. error as SetCounter) in
  let m' = setCounterClass m in
```

¹На самом деле настоящие объектно-ориентированные языки идут еще дальше. Вместо того, чтобы вычислять и хранить полную таблицу методов, каждый класс хранит только те методы, которые в нем добавлены или переопределены по сравнению с надклассом. Так что таблица методов в нашем понимании вообще не строится — при вызове метода мы просто проходим вверх по иерархии классов, начиная с класса объекта-адресата, пока не найдется определение нужного нам метода. Такой поиск во время выполнения создает нетривиальные проблемы для статического анализа типов, но здесь мы их касаться не будем.

```
(m := m';
 λ_:Unit. let r = {x=ref 1} in m' r);

▷ newSetCounter: Unit → SetCounter
```

Заметим, что первые три строки в этом определении вычисляются только один раз, когда определяется `newSetCounter`. Вычисление останавливается, дойдя до элементарной абстракции на последней строке, и эту строку можно применить сколько угодно раз при создании объектов; каждый раз будет выделяться область памяти для новой записи переменных экземпляра `r`, и таблица методов `m'` будет устанавливаться в указатель на `r`. В результате получается свежий объект.

К сожалению, переупорядочив таким образом `setCounterClass`, мы ввели в тип состояния *контравариантное* вхождение типа-состояния `CounterRep`. За это придется заплатить, когда мы попытаемся определить подкласс `setCounterClass`.

```
instrCounterClass =
  λself: Source (InstrCounterRep → InstrCounter).
  let super = setCounterClass self in
  λr: InstrCounterRep.
  {get = (super r).get,
   set = λi:Nat. (r.a:=succ(!r.a)); (super r).set i),
   inc = (super r).inc,
   accesses = λ_:Unit. !(r.a)};
```

```
▷ Ошибка: несоответствие типа параметра
```

Несоответствие возникает в определении переменной `super`, когда мы порождаем экземпляр надкласса `setCounterClass` с тем же `self`, который был передан подклассу. К сожалению, текущий `self` является ссылкой на функцию типа `InstrCounterRep → InstrCounter`, а это не подтип `CounterRep → SetCounter`, поскольку типы слева и справа от стрелок представлены местами.

С этой трудностью можно справиться, если еще раз переписать `setCounterClass`, используя теперь ограниченную квантификацию.

```
setCounterClass =
  λR<:CounterRep.
  λself: Source (R → SetCounter).
  λr: R.
  {get = λ_:Unit. !(r.x),
   set = λi:Nat. r.x:=i,
   inc = λ_:Unit. (!self r).set (succ(!self r).get unit))};

▷ setCounterClass : ∀R<:CounterRep.
                                         (Source (R → SetCounter)) → R →
                                         SetCounter
```

Этим изменением мы добиваемся того, что `setCounterClass` начинает несколько менее строго относиться к передаваемому в него параметру `self`.

Немного антропоморфизируя, можно представить себе, что предыдущая версия `setCounterClass` говорит своему окружению: «Дайте мне, пожалуйста, параметр `self`, который принимает параметром `CounterRep`, и я из него сделаю таблицу методов, которая тоже ожидает параметром `CounterRep`». Новая же версия говорит: «Пожалуйста, скажите мне, какой будет тип представления `R` у порожденного нами объекта; в этом типе, по меньшей мере, должно быть поле `x`, поскольку оно мне нужно. Потом дайте мне (ссылку на) `self`, принимающий `R` как параметр и возвращающий таблицу методов, по меньшей мере, с интерфейсом `SetCounter`, а я построю и верну еще один такой же объект».

Яснее всего эффект такого изменения виден при определении подкласса `instrCounterClass`.

```
instrCounterClass =
  λR<: InstrCounterRep .
  λself: Source(R → InstrCounter) .
  λr : R .
    let super = setCounterClass [R] self in
      {get = (super r).get,
       set = λi:Nat. (r.a:=succ(!(r.a)); (super r).set i),
       inc = (super r).inc,
       accesses = λ_:Unit. !(r.a)} ;

▷ instrCounterClass : ∀R<: InstrCounterRep .
                                         (Source (R → InstrCounter)) →
                                         R → InstrCounter
```

Этот код работает там, где предыдущий вариант ломался, потому что в выражении, связанном с переменной `super`, иначе работает правило включения: мы расширяем тип `self` и делаем его подходящим для надкласса. Раньше мы пытались показать

```
Source(InstrCounterRep → InstrCounter)
<: Source(CounterRep → SetCounter)
```

что было неверно. Теперь нам надо показать только

```
Source(R → InstrCounter) <: Source(R → SetCounter)
```

а это истинно.

Функции создания объектов для нашего нового кодирования классов очень похожи на старые варианты. Вот, например, функция создания для подкласса счетчиков с подсчетом доступа.

```
newInstrCounter =
  let m = ref (λr: InstrCounterRep. error as InstrCounter) in
  let m' = instrCounterClass [InstrCounterRep] m in
  (m := m';
   λ_:Unit. let r = {x=ref 1, a=ref 0} in m' r);

▷ newInstrCounter : Unit → InstrCounter
```

Единственное отличие состоит в том, что требуется конкретизировать `InstrCounterClass` реальным типом записи переменных экземпляра,

`InstrCounterRep`. Как и раньше, первые три строки вычисляются всего один раз, при связывании переменной `newInstrCounter`.

Наконец, приведем несколько тестов, показывающих, что наши счетчики работают как полагается:

```
ic = newInstrCounter unit;
ic.inc unit;
ic.get unit;

▷ 2 : Nat

ic.accesses unit;

▷ 1 : Nat
```

Упражнение 27.0.1 [РЕКОМЕНДУЕТСЯ ★ ★★]: *Наше новое кодирование классов опирается на ковариантность конструктора типов `Source`. Можно ли достичнуть такой же эффективности (т. е. построить правильно типизированное кодирование классов с таким же операционным поведением) в языке, в котором присутствует только ограниченная кантификация и инвариантный конструктор `Ref`?*

Глава 28

Метатеория ограниченной квантификации

В этой главе мы разрабатываем алгоритмы вычисления отношения подтиповирования и проверки типов для $F_{\leq:}$. Изучается как ядерная, так и полная версия системы; ведут они себя несколько по-разному. Некоторые свойства присутствуют в обоих вариантах, но в полном доказать их сложнее; другие характеристики в полной $F_{\leq:}$ попросту утрачиваются — такова цена, которую приходится платить за большую выразительность этой системы.

Сначала, в §28.1 и §28.2, мы представим алгоритм проверки типов, работающий в обеих системах. Затем мы рассмотрим проверку подтиповирования, сначала для ядерной системы в §28.3, а потом для полной в §28.4. В §28.5 продолжается обсуждение подтиповирования в полной $F_{\leq:}$, и особое внимание уделяется тому удивительному обстоятельству, что отношение подтиповирования неразрешимо. В §28.6 мы покажем, что в ядерной $F_{\leq:}$ имеются пересечения и объединения, а в полной их нет. §28.7 затрагивает некоторые вопросы, связанные с ограниченными экзистенциальными типами, а в §28.8 мы рассмотрим, к чему приводит добавление наименьшего типа `Bot`.

28.1. Выявление

В алгоритме проверки типов для простого типизированного лямбда-исчисления с подтипами из §16.2 ключевой идеей было вычислять *наименьший тип* (minimal type) для каждого терма, исходя из наименьших типов его подтермов. Ту же самую базовую идею можно использовать и для $F_{\leq:}$, однако нужно принять во внимание небольшую сложность, которая возникает из-за наличия в системе типовых переменных. Рассмотрим терм

```
f = λx<:Nat → Nat. λy:x. y 5;
```

В этой главе изучается чистая система $F_{\leq:}$ (рис. 26.1). Соответствующая реализация на OCaml называется `purefsub`; реализация `fullfsub` включает также экзистенциальные типы (24.1) и некоторые расширения из главы 11.

$\rightarrow \forall <: \text{Top}$

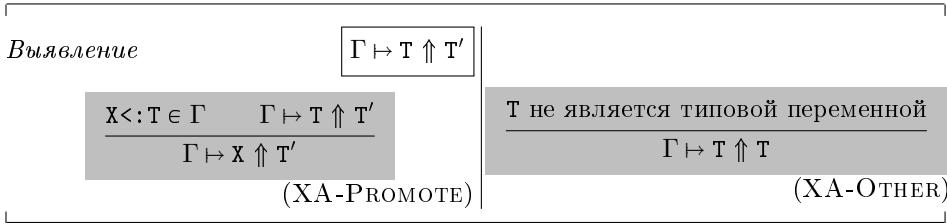


Рис. 28.1. Алгоритм выявления для $F_<$.

$\triangleright f : \forall X <: \text{Nat} \rightarrow \text{Nat}. \quad X \rightarrow \text{Nat}$

Ясно, что этот терм правильно типизирован, поскольку тип переменной u в применении $u 5$ может быть расширен до $\text{Nat} \rightarrow \text{Nat}$ по правилу T-Sub. Однако *наименьший* тип u равен X , и это не функциональный тип. Чтобы определить наименьший тип всего терма-применения, нужно найти наименьший функциональный тип для u — т. е. наименьший функциональный тип, являющийся надтиром типовой переменной X . Неудивительно, что такой тип можно найти, *расширяя* (promoting) наименьший тип u , пока он не превратится в нечто отличное от типовой переменной.

Мы используем формальную запись $\Gamma \vdash S \uparrow T$ (произносится как « S выявляется (exposes) как T в контексте Γ »), что означает « T — наименьший надтип S , не являющийся переменной». Выявление определяется через циклическое расширение типовых переменных, как показано на рис. 28.1.

Нетрудно убедиться, что эти правила дают всюду определенную функцию. Более того, результатом выявления типа всегда будет наименьший надтип, отличный от переменной. Например, если $\Gamma = X <: \text{Top}, Y <: \text{Nat} \rightarrow \text{Nat}, Z <: Y, W <: Z$, то

$$\begin{array}{lll} \Gamma \vdash \text{Top} \uparrow \text{Top} & \Gamma \vdash Y \uparrow \text{Nat} \rightarrow \text{Nat} & \Gamma \vdash W \uparrow \text{Nat} \rightarrow \text{Nat} \\ \Gamma \vdash X \uparrow \text{Top} & \Gamma \vdash Z \uparrow \text{Nat} \rightarrow \text{Nat} & \end{array}$$

Основные свойства выявления можно описать следующим образом.

Лемма 28.1.1 [Выявление]: Допустим, $\Gamma \vdash S \uparrow T$. Тогда

1. $\Gamma \vdash S <: T$
2. Если $\Gamma \vdash S <: U$, где U — не переменная, то $\Gamma \vdash T <: U$.

Доказательство: Часть (1) доказывается индукцией по деревьям вывода $\Gamma \vdash S \uparrow T$, часть (2) индукцией по деревьям вывода $\Gamma \vdash S <: U$.

28.2. Минимальная типизация

Алгоритм вычисления наименьших типов строится по тем же принципам, что и для простого типизированного лямбда-исчисления с подтипами, но с одной дополнительной деталью: когда мы проверяем тип в терме-применении,

$\rightarrow \forall <: \text{Top}$ Расширяет $\lambda <:$ (16.3)

Алгоритмическая типизация		
$\boxed{\Gamma \vdash t : T}$		
$\frac{x:T \in \Gamma}{\Gamma \vdash x : T}$	(TA-VAR)	$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X <: T_1 . t_2 : \forall X <: T_1 . T_2}$ (TA-TABs)
$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1 . t_2 : T_1 \rightarrow T_2}$	(TA-ABS)	$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash T_1 \uparrow \forall X <: T_{11} . T_{12}}{\Gamma \vdash t_1 [T_2] : [X \mapsto T_2] T_{12}}$ (TA-TAPP)
$\frac{\Gamma \vdash t_1 : T_1 \quad \boxed{\Gamma \vdash T_1 \uparrow (T_{11} \rightarrow T_{12})} \quad \Gamma \vdash t_2 : T_2 \quad \boxed{\Gamma \vdash T_2 <: T_{11}}}{\Gamma \vdash t_1 t_2 : T_{12}}$	(TA-APP)	

Рис. 28.2. Алгоритмическая типизация для $F_{<:}$.

мы сначала вычисляем наименьший тип левой части, а затем *выявляем* его, получая функциональный тип, как показано на рис. 28.2. Если же при выявлении левой части применения не получается функционального типа, то правило TA-APP оказывается неприменимо, и терм типизирован неверно. Аналогично мы проверяем типы в применении типа: выявляем левую часть и надеемся при этом получить кванторный тип.

Доказательство корректности и полноты этого алгоритма по отношению к исходным правилам типизации не представляют труда. Мы приводим доказательство для ядерной $F_{<:}$. (рассуждение для полной $F_{<:}$ строится аналогично; ср. упражнение 28.2.3).

Теорема 28.2.1 [Минимальная типизация]:

1. Если $\Gamma \vdash t : T$, то $\Gamma \vdash t : T$.
2. Если $\Gamma \vdash t : T$, то $\Gamma \vdash t : M$, причем $\Gamma \vdash M <: T$.

Доказательство: Часть (1) представляет собой несложную индукцию по алгоритмическим выводам, с использованием части (1) леммы 28.1.1 для вариантов с применением. В части (2) проводится индукция по дереву вывода $\Gamma \vdash t : T$, с разбором вариантов последнего правила в выводе. Наиболее интересны варианты Т-APP и Т-TAPP.

Вариант Т-VAR: $t = x \quad x:T \in \Gamma$

По правилу TA-VAR, $\Gamma \vdash x : T$. По правилу S-REFL, $\Gamma \vdash T <: T$.

Вариант Т-ABS: $t = \lambda x:T_1 . t_2 \quad \Gamma, x:T_1 \vdash t_2 : T_2 \quad T = T_1 \rightarrow T_2$

Согласно предположению индукции, $\Gamma, x:T_1 \vdash t_2 : M_2$ для некоторого M_2 ,

причем $\Gamma, x:T_1 \vdash M_2 <: T_2$ — т. е. $\Gamma \vdash M_2 <: T_2$, поскольку отношение подтипования не зависит от связываний термовых переменных в контексте (лемма 26.4.4). По правилу ТА-ABS, $\Gamma \vdash t : T_1 \rightarrow M_2$. По правилам S-REFL и S-ARROW, $\Gamma \vdash T_1 \rightarrow M_2 <: T_1 \rightarrow T_2$.

Вариант Т-APP: $t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad T = T_{12} \quad \Gamma \vdash t_2 : T_{11}$
 Согласно предположению индукции, имеем $\Gamma \vdash t_1 : M_1$ и $\Gamma \vdash t_2 : M_2$, причем $\Gamma \vdash M_1 <: T_{11} \rightarrow T_{12}$ и $\Gamma \vdash M_2 <: T_{11}$. Пусть N_1 — наименьший надтип M_1 , не являющийся переменной, т. е. $\Gamma \vdash M_1 \uparrow N_1$. Согласно части (2) леммы 28.1.1, $\Gamma \vdash M_1 <: T_{11} \rightarrow T_{12}$. Поскольку мы знаем, что N_1 — не переменная, лемма об инверсии для отношения подтипования (26.4.10) сообщает нам, что $N_1 = N_{11} \rightarrow N_{12}$, причем $\Gamma \vdash T_{11} <: N_{11}$ и $\Gamma \vdash N_{12} <: T_{12}$. По transitivности, $\Gamma \vdash M_2 <: N_{11}$, так что применимо правило ТА-APP, и оно дает нам $\Gamma \vdash t_1 \ t_2 : N_{12}$. Все требования теоремы при этом оказываются соблюдены.

Вариант Т-TABS: $t = \lambda X <: T_1 . t_2 \quad \Gamma, X <: T_1 \vdash t_2 : T_2 \quad T = \forall X <: T_1 . T_2$
 По предположению индукции, $\Gamma, X <: T_1 \vdash t_2 : M_2$ для некоторого M_2 , причем $\Gamma, X <: T_1 \vdash M_2 <: T_2$. По правилу ТА-TABS, $\Gamma \vdash t : \forall X <: T_1 . M_2$. По правилу S-ALL, $\Gamma \vdash \forall X <: T_1 . M_2 <: \forall X <: T_1 . T_2$.

Вариант Т-TAPP: $t = t_1 [T_2] \quad \Gamma \vdash t_1 : \forall X <: T_{11} . T_{12}$
 $T = [X \mapsto T_2] T_{12} \quad \Gamma \vdash T_2 <: T_{11}$

Согласно предположению индукции, имеем $\Gamma \vdash t_1 : M_1$, причем $\Gamma \vdash M_1 <: \forall X <: T_{11} . T_{12}$. Пусть N_1 — наименьший надтип M_1 , не являющейся переменной, т. е. $\Gamma \vdash M_1 \uparrow N_1$. По лемме о выявлении (28.1.1), $\Gamma \vdash N_1 <: \forall X <: T_{11} . T_{12}$. Но мы знаем, что N_1 — не переменная, так что по лемме об инверсии для отношения подтипования (26.4.10), имеем $N_1 = \forall X <: T_{11} . N_{12}$, причем $\Gamma, X <: T_{11} \vdash N_{12} <: T_{12}$. Правило ТА-TAPP дает нам $\Gamma \vdash t_1 [T_2] : [X \mapsto T_2] N_{12}$, а поскольку отношение подтипования сохраняется при подстановке (лемма 26.4.8), $\Gamma \vdash [X \mapsto T_2] N_{12} <: [X \mapsto T_2] T_{12} = T$.

Вариант Т-SUB: $\Gamma \vdash t : S \quad \Gamma \vdash S <: T$

Согласно предположению индукции, $\Gamma \vdash t : M$, причем $\Gamma \vdash M <: S$. По transitivности, $\Gamma \vdash M <: T$.

Следствие 28.2.2 [РАЗРЕШИМОСТЬ ТИПИЗАЦИИ]: Отношение типизации для ядерной $F_{<}$ разрешимо, если имеется разрешающая процедура для отношения подтипования.

Доказательство: Для любых Γ и t можно проверить, существует ли какой-либо тип T , такой что $\Gamma \vdash t : T$, породив при помощи алгоритмических правил доказательство $\Gamma \vdash t : T$. Если доказательство прошло успешно, то полученный T является также типом для t согласно исходному отношению типизации, по части (1) теоремы 28.2.1. В противном случае, из части (2) теоремы 28.2.1 следует, что t не имеет типа в исходном отношении типизации. Наконец, заметим, что алгоритмические правила типизации соответствуют всегда завершающемуся алгоритму, поскольку они управляются синтаксисом (каждому данному терму t применимо не более одного правила) и всегда уменьшают размер t в направлении снизу вверх.

Упражнение 28.2.3 [★★]: Как нужно изменить вышеприведенное доказательство, чтобы оно работало с полной $F_{<:}$?

28.3. Подтипы в ядерной $F_{<:}$

В §16.1 мы отмечали, что декларативное отношение подтипов для простого типизированного лямбда-исчисления с подтипами не управляетяется синтаксисом, т. е. его невозможно прочесть как алгоритм определения подтипов для, по двум причинам: (1) заключения правил S-REFL и S-TRANS перекрываются с другими правилами (так что, если мы читаем правила снизу вверх, мы не знаем, которое из правил следует применить), и (2) в предпосылках S-TRANS упоминается метапеременная, которая не встречается в его заключении (и наивному алгоритму пришлось бы как-то «угадывать» ее значение). Мы видели, что можно справиться с этими проблемами, просто исключив неудобные для алгоритмизации правила из системы. Однако для этого нам сначала потребовалось немного подправить систему, сведя три раздельных правила подтипов для записей в одно.

Для ядерной $F_{<:}$ ситуация аналогична. Неудобными правилами снова являются S-REFL и S-TRANS, и мы получаем алгоритм, устранив эти правила и изменяя остающиеся правила так, что в результате они берут на себя обработку тех случаев, которые раньше обрабатывались устранимыми правилами.

В простом типизированном лямбда-исчислении с подтипами не было случаев, которые требовали бы наличия правила рефлексивности — его можно было просто отбросить, не повлияв на множество выводимых утверждений о подтиповании (лемма 16.1.2, часть 1). Напротив, в $F_{<:}$ утверждения л подтипованием вида $\Gamma \vdash X <: X$ доказываются только через рефлексивность. Так что, когда мы удаляем полное правило рефлексивности, вместо него следует добавить ограниченную аксиому рефлексивности, касающуюся только переменных.

$$\Gamma \vdash X <: X$$

Аналогично, чтобы избавиться от правила S-TRANS, следует сначала понять, какие случаи его использования неустранимы. Здесь интерес представляет взаимодействие с правилом S-TVAR, которое позволяет использовать предположения о типовых переменных при выводе утверждений о подтиповании. Например, если $\Gamma = W <: \text{Top}, X <: W, Y <: X, Z <: Y$, то утверждение $\Gamma \vdash Z <: W$ невозможно вывести, если в системе отсутствует правило S-TRANS. В общем случае экземпляр S-TRANS, где левый подвывод является экземпляром аксиомы S-TVAR, как в

$$\frac{\frac{Z <: Y \in \Gamma}{\Gamma \vdash Z <: Y} \text{ (S-TVAR)} \quad : \quad \frac{}{\Gamma \vdash Y <: W}}{\Gamma \vdash Z <: W} \text{ (S-TRANS)}$$

неустраним.

К счастью, выводы такого вида — единственный важный неустранимый класс случаев использования транзитивности при выводе подтипования.

$\rightarrow \forall <: \text{Top}$

Расширяет $\lambda_{<:} (16.2)$

Алгоритмическое подтиповование	
$\Gamma \vdash S <: T$	
$\boxed{\Gamma \vdash S <: \text{Top}} \quad (\text{SA-TOP})$	
$\Gamma \vdash X <: X \quad (\text{SA-REFL-TVAR})$	
$\frac{X <: U \in \Gamma \quad \Gamma \vdash U <: T}{\Gamma \vdash X <: T} \quad (\text{SA-TRANS-TVAR})$	
	$\frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{SA-ARROW})$ $\frac{\Gamma, X <: U_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: U_1 . S_2 <: \forall X <: U_1 . T_2} \quad (\text{SA-ALL})$

Рис. 28.3. Алгоритмическое отношение подтиповования для ядерной $F_{<:}$

Это наблюдение можно выразить точно, введя новое правило подтиповования

$$\frac{X <: U \in \Gamma \quad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

которое охватывает в точности эту форму поиска переменной, за которым следует применение транзитивности, и показывает, что замена правил транзитивности и поиска переменной этим правилом не влияет на множество выводимых утверждений о подтиповании.

Эти изменения приводят нас к алгоритмическому отношению подтиповования для ядерной $F_{<:}$, изображенном на рис. 28.3. Мы добавляем стрелку к концу символа «штопора» в алгоритмических утверждениях о типизации, чтобы отличить их от исходной формы утверждений о типизации, когда речь в тексте идет об обеих разновидностях.

Тот факт, что новых правил SA-REFL-TVAR и SA-TRANS-TVAR достаточно для замены старых правил рефлексивности и транзитивности, подтверждается следующими двумя леммами.

Лемма 28.3.1 [Рефлексивность алгоритмического отношения подтиповования]: Для каждого типа T и контекста Γ можно доказать, что $\Gamma \vdash T <: T$.

Доказательство: Индукция по T .

Лемма 28.3.2 [Транзитивность алгоритмического отношения подтипования]: Если $\Gamma \vdash S <: Q$ и $\Gamma \vdash Q <: T$, то $\Gamma \vdash S <: T$.

Доказательство: Индукция по сумме размеров двух деревьев вывода. Имея два подвывода, мы рассматриваем последние правила обоих этих подвыводов.

Если правый подывод является экземпляром SA-TOP, то доказательство закончено, поскольку $\Gamma \vdash S <: \text{Top}$ по правилу SA-TOP. Если левый

подвывод — экземпляром SA-ТОР, то $Q = \text{Top}$, и, рассматривая алгоритмические правила, мы видим, что и правый подвывод обязан быть экземпляром SA-ТОР.

Если какой-либо подвывод является экземпляром SA-REFL-TVAR, то, опять же, все доказано, поскольку другое поддерево будет в точности являться желаемым результатом.

Если левый подвывод завершается экземпляром SA-TRANS-TVAR, то $S = Y$, причем $Y <: U \in \Gamma$, и у нас есть подвывод с заключением $\Gamma \vdash U <: Q$. Согласно предположению индукции, $\Gamma \vdash U <: T$, и, снова по правилу SA-TRANS-TVAR, имеем $\Gamma \vdash Y <: T$, что и требуется.

Если левое поддерево заканчивается экземпляром SA-ARROW, имеем $S = S_1 \rightarrow S_2$ и $Q = Q_1 \rightarrow Q_2$, с подвыводами $\Gamma \vdash Q_1 <: S_1$ и $\Gamma \vdash S_2 <: Q_2$. Однако, поскольку мы уже рассмотрели вариант, в котором правый подвывод представляет собой SA-ТОР, единственная оставшаяся возможность состоит в том, что этот вывод также заканчивается на SA-ARROW, а значит, имеем $T = T_1 \rightarrow T_2$ и еще два подвывода $\Gamma \vdash T_1 <: Q_1$ и $\Gamma \vdash Q_2 <: T_2$. Теперь дважды применяем предположение индукции, получая $\Gamma \vdash T_1 <: S_1$ и $\Gamma \vdash S_2 <: T_2$. Наконец, SA-ARROW дает нам $\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2$, что и требуется.

В случае, когда левый подвывод заканчивается экземпляром SA-ALL, рассуждение проходит аналогично. Имеем $S = \forall X <: U_1 . S_2$ и $Q = \forall X <: U_1 . Q_2$, а также подвывод $\Gamma, X <: U_1 \vdash S_2 <: Q_2$. Опять же, поскольку мы уже рассмотрели вариант, когда правый подвывод является экземпляром SA-ТОР, он должен заканчиваться на SA-ALL; так что $T = \forall X <: U_1 . T_2$, причем имеется подвывод $\Gamma, X <: U_1 \vdash Q_2 <: T_2$. Из предположения индукции получаем $\Gamma, X <: U_1 \vdash S_2 <: T_2$, и, по правилу SA-ALL, $\Gamma \vdash \forall X <: U_1 . S_2 <: \forall X <: U_1 . T_2$.

Теорема 28.3.3 [КОРРЕКТНОСТЬ И ПОЛНОТА АЛГОРИТМИЧЕСКОГО ОТНОШЕНИЯ ПОДТИПИРОВАНИЯ] : $\Gamma \vdash S <: T$ тогда и только тогда, когда $\Gamma \vdash S <: T$.

Доказательство: В обоих направлениях проводится индукция по деревьям вывода. Корректность (\Leftarrow) не представляет труда. При доказательстве полноты (\Rightarrow) используются леммы 28.3.1 и 28.3.2.

Наконец, требуется убедиться в том, что правила для подтиповования определяют *всюду определенный* (total) алгоритм — т. е. алгоритм, завершающийся при любых входных данных. Мы сделаем это, присваивая каждому утверждению о подтиповании вес, и показывая, что каждое алгоритмическое правило имеет заключение со строго большим весом, чем у предпосылок.

Определение 28.3.4 Вес типа T в контексте Γ , который обозначается как $\text{weight}_{\Gamma}(T)$, определяется так:

$$\begin{aligned} \text{weight}_{\Gamma}(X) &= \text{weight}_{\Gamma}(U) + 1 \quad \text{если } \Gamma = \Gamma_1, X <: U, \Gamma_2 \\ \text{weight}_{\Gamma}(\text{Top}) &= 1 \\ \text{weight}_{\Gamma}(T_1 \rightarrow T_2) &= \text{weight}_{\Gamma}(T_1) + \text{weight}_{\Gamma}(T_2) + 1 \\ \text{weight}_{\Gamma}(\forall X <: T_1 . T_2) &= \text{weight}_{\Gamma, X <: T_1}(T_2) + 1 \end{aligned}$$

Вес утверждения о подтиповании $\Gamma \vdash S <: T$ есть сумма весов S и T в контексте Γ .

$\rightarrow \forall <: \text{Top}$ полная

Расширяет 28.3

Алгоритмическое подтиповование	
$\Gamma \mapsto S <: T$	
$\Gamma \mapsto S <: \text{Top}$	(SA-TOP)
$\Gamma \mapsto X <: X$	(SA-REFL-TVAR)
$X <: U \in \Gamma \quad \Gamma \mapsto U <: T$	$\frac{}{\Gamma \mapsto X <: T}$ (SA-TRANS-TVAR)
	$\frac{\Gamma \mapsto T_1 <: S_1 \quad \Gamma \mapsto S_2 <: T_2}{\Gamma \mapsto S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$ (SA-ARROW)
	$\frac{\Gamma \mapsto T_1 <: S_1 \quad \Gamma, X <: T_1 \mapsto S_2 <: T_2}{\Gamma \mapsto \forall X <: S_1 . S_2 <: \forall X <: T_1 . T_2}$ (SA-ALL)

Рис. 28.4. Алгоритмическое отношение подтиповования для полной $F_{<:}$.

Теорема 28.3.5 Алгоритм проверки подтиповования завершается при любом вводе.

Доказательство: Вес заключения в любом экземпляре алгоритмического правила подтиповования всегда строго больше, чем вес каждой из предпосылок.

Следствие 28.3.6 Отношение подтиповования в ядерной $F_{<:}$ разрешимо.

28.4. Подтипы в полной $F_{<:}$:

Алгоритм проверки подтиповования для полной $F_{<:}$, приведенный на рис. 28.4, почти такой же, как для ядерной $F_{<:}$; единственное изменение заключается в замене SA-ALL на более гибкий вариант. Как и в случае ядерной $F_{<:}$, корректность и полнота этого алгоритмического отношения по сравнению с исходным отношением подтиповования прямо следуют из рефлексивности и транзитивности алгоритмического отношения.

Рассуждение для рефлексивности остается в точности таким же, как раньше, однако доказательство транзитивности оказывается несколько более тонким. Чтобы понять, почему это так, вспомним доказательство транзитивности для ядерной $F_{<:}$ из предыдущего раздела (лемма 28.3.2). Там идея состояла в том, чтобы взять два дерева вывода для отношения подтиповования, завершающиеся утверждениями $\Gamma \vdash S <: Q$ и $\Gamma \vdash Q <: T$, и показать, как представить и пересобрать их поддеревья, получая при этом вывод $\Gamma \vdash S <: T$, не используя правило транзитивности и предполагая (в качестве предположения индукции), что то же самое можно проделать для выводов меньшего размера. Предположим теперь, что у нас есть два подвыхода, завершающиеся новым

правилом SA-ALL:

$$\frac{\vdots \quad \vdots}{\Gamma \vdash Q_1 <: S_1} \quad \frac{\vdots \quad \vdots}{\Gamma, X <: Q_1 \vdash S_2 <: Q_2} \quad \frac{\vdots \quad \vdots}{\Gamma \vdash T_1 <: Q_1} \quad \frac{\vdots \quad \vdots}{\Gamma, X <: T_1 \vdash Q_2 <: T_2}$$

$$\Gamma \vdash \forall X <: S_1 . S_2 <: \forall X <: Q_1 . Q_2 \quad \Gamma \vdash \forall X <: Q_1 . Q_2 <: \forall X <: T_1 . T_2$$

Следуя схеме предыдущего доказательства, мы хотели бы воспользоваться предположением индукции, чтобы объединить левый и правый подвыводы и получить единственный экземпляр SA-ALL с заключением $\Gamma \vdash \forall X <: S_1 . S_2 <: \forall X <: T_1 . T_2$. Для левых подвыводов никаких сложностей нет; предположение индукции дает нам вывод $\Gamma \vdash T_1 <: S_1$, не использующий транзитивности. Однако для правых подвыводов предположение индукции неприменимо, поскольку контексты подвыводов различаются: верхняя граница для X в одном из них равна Q_1 , а в другом — T_1 .

К счастью, мы знаем, как *сделать* контексты одинаковыми: свойство *сужения* (narrowing) из главы 26 (лемма 26.4.5) говорит, что истинное утверждение о подтиповании остается истинным, если мы заменяем граничный тип в контексте одним из его подтипов. Так что, казалось бы, можно просто сузить подывод $\Gamma, X <: Q_1 \vdash S_2 <: Q_2$ до $\Gamma, X <: T_1 \vdash S_2 <: Q_2$, и таким образом разрешить использование предположения индукции.

Тут, однако, необходима известная осторожность. Лемма 26.4.5 говорит, что можно взять произвольный вывод и породить вывод с суженным заключением, однако она *не гарантирует*, что размер нового вывода будет таким же, как у старого. В самом деле, рассмотрев доказательство этой леммы, мы увидим, что, как правило, сужение порождает вывод большего размера, чем исходный, поскольку всюду, где аксиома S-TVAR используется для поиска сужаемой переменной, вставляется копия произвольно большого вывода. Более того, эта операция вставки приводит к порождению новых экземпляров правила транзитивности, а именно отсутствие необходимости в этом правиле в нашей текущей системе мы пытаемся доказать.

Чтобы справиться с этими сложностями, мы доказываем транзитивность и сужение *совместно*, с предположением индукции, основанном на размере промежуточного типа Q для свойства транзитивности и размере исходного типа-границы Q для свойства сужения.

Прежде чем начать основное доказательство, мы приводим несложную лемму, утверждающую, что порядок добавления новых связываний типовых переменных в контекст не влияет на верность выводимых утверждений о подтиповании.

Лемма 28.4.1 [ПЕРЕСТАНОВКА И ОСЛАБЛЕНИЕ]:

1. Предположим, что Δ является правильно сформированной перестановкой Γ (ср. 26.4.1). Если $\Gamma \vdash S <: T$, то $\Delta \vdash S <: T$.
2. Если $\Gamma \vdash S <: T$ и $dom(\Delta) \cap dom(\Gamma) = \emptyset$, то $\Gamma, \Delta \vdash S <: T$.

Доказательство: Прямолинейное доказательство по индукции. Часть (1) используется в варианте SA-ALL части (2).

Лемма 28.4.2 [Транзитивность и сужение для полной F_{\leq}]:

1. Если $\Gamma \vdash S <: Q$ и $\Gamma \vdash Q <: T$, то $\Gamma \vdash S <: T$.
2. Если $\Gamma, X <: Q, \Delta \vdash M <: N$ и $\Gamma \vdash P <: Q$, то $\Gamma, X <: P, \Delta \vdash M <: N$.

Доказательство: Обе части леммы доказываются одновременно, индукцией по размеру Q . На каждом шаге индукции в доказательстве части (2) мы предполагаем, что часть (1) уже доказана для нашего Q ; часть (1) использует часть (2) только для типов Q строго меньшего размера.

1. Проводим внутреннюю индукцию по размеру первого данного вывода, и анализируем последнее правило в обоих подвыводах. Все варианты, кроме одного, совпадают с доказательством леммы 28.3.2; различие касается случая SA-ALL.

Если правый подывод является экземпляром SA-TOP, то доказательство закончено, поскольку $\Gamma \vdash S <: \text{Top}$ по правилу SA-TOP. Если левый подывод — экземпляр SA-TOP, то $Q = \text{Top}$ и, рассматривая алгоритмические правила, мы видим, что и правый подывод тогда обязан быть экземпляром SA-TOP. Если какой-либо из подыводов является экземпляром SA-REFL-TVAR, то, опять же, лемма доказана, поскольку другое поддерево будет в точности желаемым результатом.

Если левое поддерево заканчивается экземпляром правила SA-TRANS-TVAR, то имеем $S = Y$, причем $Y <: U \in \Gamma$, и это выражение есть подывод утверждения $\Gamma \vdash U : Q$. По внутреннему предположению индукции, $\Gamma \vdash U <: T$, и, снова по SA-TRANS-TVAR, $\Gamma \vdash Y <: T$, что нам и требуется.

Если левое поддерево заканчивается экземпляром правила SA-ARROW или SA-ALL, то, поскольку вариант с правилом SA-TOP в качестве правого подывода уже рассмотрен, правый подывод должен завершаться тем же правилом, что и левый. Если это правило — SA-ARROW, то имеем $S = S_1 \rightarrow S_2$, $Q = Q_1 \rightarrow Q_2$ и $T = T_1 \rightarrow T_2$ с подвыводами $\Gamma \vdash Q_1 <: S_1$, $\Gamma \vdash S_2 <: Q_2$, $\Gamma \vdash T_1 <: Q_1$ и $\Gamma \vdash Q_2 <: T_2$. Применим часть (1) внешнего предположения индукции дважды (поскольку как Q_1 , так и Q_2 меньше по размеру, чем Q) и получаем $\Gamma \vdash T_1 <: S_1$ и $\Gamma \vdash S_2 <: T_2$. Наконец, с помощью правила SA-ARROW получаем $\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2$.

В случае, когда оба подывода заканчиваются на SA-ALL, имеем $S = \forall X <: S_1 . S_2$, $Q = \forall X <: Q_1 . Q_2$ и $T = \forall X <: T_1 . T_2$, и есть подыводы

$$\begin{aligned} \Gamma \vdash Q_1 &<: S_1 & \Gamma, X <: Q_1 \vdash S_2 &<: Q_2 \\ \Gamma \vdash T_1 &<: Q_1 & \Gamma, X <: T_1 \vdash Q_2 &<: T_2 \end{aligned}$$

Согласно части (1) внешнего предположения индукции (поскольку Q_1 меньше по размеру, чем Q), мы можем обединить два подывода для ограничений и получить $\Gamma \vdash T_1 <: S_1$. Для тел кванторов приходится приложить немного больше усилий, поскольку контексты не совсем совпадают. Используем часть (2) внешнего предположения индукции

(поскольку Q_2 меньше, чем Q) и сужаем ограничение для X в подвыводе $\Gamma, X <: Q_1 \mapsto S_2 <: Q_2$, так что получается $\Gamma, X <: T_1 \mapsto S_2 <: Q_2$. Теперь применима часть (1) внешнего предположения индукции (поскольку Q_2 меньше, чем Q); она дает нам $\Gamma, X <: T_1 \mapsto S_2 <: T_2$. Наконец, по правилу SA-ALL, $\Gamma \mapsto \forall X <: S_1 . S_2 <: \forall X <: T_1 . T_2$.

2. Снова проведем внутреннюю индукцию по размеру первого данного подвывода, рассматривая варианты последнего правила в нем. В большинстве вариантов нужно всего лишь очевидным образом использовать внутреннее предположение индукции. Интерес представляет вариант SA-TRANS-TVAR с $M = X$, где в качестве подвывода мы имеем $\Gamma, X <: Q, \Delta \mapsto Q <: N$. Применяя внутреннее предположение индукции к этому подвыводу, получаем $\Gamma, X <: P, \Delta \mapsto Q <: N$. Кроме того, через ослабление (лемма 28.4.1, часть 2) второго данного вывода получаем $\Gamma, X <: P, \Delta \mapsto P <: Q$. Теперь через часть (1) внешнего предположения индукции (с тем же самым Q) имеем $\Gamma, X <: P, \Delta \mapsto P <: N$. Наконец, применяем правило SA-TRANS-TVAR и получаем $\Gamma, X <: P, \Delta \mapsto X <: N$, что и требуется.

Упражнение 28.4.3 [★ ★ ★ →]: Есть еще один осмысленный вариант правила подтипования для кванторов, несколько более гибкий, чем правило ядерной $F_{<:}$, но существенно более слабый, чем правило полной $F_{<:}$:

$$\frac{\Gamma \vdash S_1 <: T_1 \quad \Gamma \vdash T_1 <: S_1 \quad \Gamma, X <: T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X <: S_1 . S_2 <: \forall X <: T_1 . T_2} \quad (\text{S-ALL})$$

Это правило близко к варианту ядерной $F_{<:}$, но требует не синтаксического совпадения границ двух кванторов, а только их эквивалентности — каждый из них должен быть подтиповом другого. Разница между ядерным правилом и этим проявляется только тогда, когда мы обогащаем язык какой-нибудь конструкцией, правила подтипования которой порождают нетривиальные классы эквивалентности между типами, например, записями. Скажем, в чистой ядерной $F_{<:}$ с записями тип $\forall X <: \{a : \text{Top}, b : \text{Top}\}.X$ не будет подтиповом $\forall X <: \{b : \text{Top}, a : \text{Top}\}.X$, а в системе с предлагаемым правилом — будет. Разрешимо ли подтиповование в системе с таким правилом?

28.5. Неразрешимость полной $F_{<:}$

В предыдущем разделе мы установили, что алгоритмические правила подтипования для полной $F_{<:}$ корректны и полны — т. е., наименьшее отношение, замкнутое относительно этих правил, содержит те же самые утверждения, что и наименьшее отношение, замкнутое относительно исходных декларативных правил. Остается нерешенным вопрос о том, завершается ли алгоритм, реализующий эти правила, при всех возможных входах. К сожалению, это не так. Когда этот факт был обнаружен, многие были удивлены.

Упражнение 28.5.1 [★]: Если алгоритмические правила для полной $F_<$, не определяют алгоритм, который всегда завершается, то, очевидно, доказательство завершения для ядерной $F_<$, невозможно перенести на правила полной системы. Где именно оно ломается?

Вот пример, найденный Гелли (Ghelli, 1995), который приводит к зацикливанию алгоритма проверки подтипования. Сначала определим следующее сокращение:

$$\neg S \stackrel{\text{def}}{=} \forall X <: S . X.$$

Ключевое свойство оператора \neg состоит в том, что он позволяет обменивать местами стороны утверждений о подтиповании.

Утверждение 28.5.2 $\Gamma \vdash \neg S <: \neg T$ тогда и только тогда, когда $\Gamma \vdash S <: T$.
Доказательство: УПРАЖНЕНИЕ [★★→].

Определим теперь тип T следующим образом:

$$T = \forall X <: \text{Top} . \neg(\forall Y <: X . \neg Y)$$

Если с помощью алгоритмических правил подтипования мы попробуем снизу вверх построить дерево вывода для утверждения

$$X_0 <: T \quad \mapsto \quad X_0 <: \forall X_1 <: X_0 . \neg X_1$$

то мы получим бесконечную последовательность всё возрастающих подцелей:

$$\begin{array}{lll} X_0 <: T & \mapsto & X_0 \\ X_0 <: T & \mapsto & \forall X_1 <: \text{Top} . \neg(\forall X_2 <: X_1 . \neg X_2) \\ X_0 <: T, X_1 <: X_0 & \mapsto & \neg(\forall X_2 <: X_1 . \neg X_2) \\ X_0 <: T, X_1 <: X_0 & \mapsto & X_1 \\ X_0 <: T, X_1 <: X_0 & \mapsto & X_0 \end{array}$$

и т. д.

Шаги переименования, требуемые для сохранения корректности контекста, здесь производятся без специального указания, и имена переменных выбираются так, чтобы лучше была видна схема бесконечного регресса. Основная хитрость состоит в «смене границ», которая происходит, например, между второй и третьей строкой, где граница для X_1 в левой части, которая в строке 2 была равна Тор, в строке 3 становится равной X_0 . Поскольку в строке 2 вся левая сторона сама по себе является верхней границей для X_0 , такая смена границ приводит к циклическому процессу, в каждой итерации которого контекст содержит все более и более длинные цепочки переменных. (Мы предупреждаем читателя, что в этом примере не стоит искать семантического смысла; в частности, $\neg T$ представляет собой отрицание только синтаксически.)

Хуже того, не только один определенный алгоритм зацикливается на некоторых входных данных, но можно также показать (Pierce, 1994), что *не существует* алгоритма, корректного и полного относительно полной $F_<$, и завершающегося на всех входных данных. Доказательство этой теоремы слишком длинно для данной книги. Однако, чтобы ощутить её общую идею, мы приведем еще один пример.

Определение 28.5.3 Положительные (*positive*) и отрицательные (*negative*) вхождения в типе T определяются следующим образом. Сам T является положительным вхождением в T . Если $T_1 \rightarrow T_2$ является положительным (или, соответственно, отрицательным) вхождением, то T_1 является отрицательным (или, соотв., положительным) вхождением, а T_2 является положительным (или, соотв., отрицательным) вхождением. Если $\forall X <: T_1 . T_2$ — положительное (или, соотв., отрицательное) вхождение, то T_1 — отрицательное (соотв. положительное), а T_2 — положительное (или, соотв., отрицательное) вхождение. Положительные и отрицательные вхождения в утверждение о подтиповании $\Gamma \vdash S <: T$ определяются так: тип S и границы типовых переменных в Γ отрицательны, а тип T положителен.

Термины «положительный» и «отрицательный» происходят из логики. Согласно широко известному *соотношению Карри-Говарда* (Curry-Howard correspondence) между пропозициями и типами (§9.4), тип $S \rightarrow T$ соответствует логическому утверждению $S \Rightarrow T$, которое, по определению логической импликации, эквивалентно $\neg S \vee T$. Подутверждение S , очевидно, находится здесь в «отрицательной» позиции — а именно, внутри нечетного количества отрицаний — тогда и только тогда, когда вся импликация расположена внутри четного количества отрицаний. Заметим, что положительное вхождение в T соответствует отрицательному вхождению в $\neg T$.

Утверждение 28.5.4 Если X входит в S только положительно, а в T — только отрицательно, то $X <: U \vdash S <: T$ тогда и только тогда, когда $\vdash [X \mapsto U]S <: [X \mapsto U]T$.

Доказательство: УПРАЖНЕНИЕ [$\star\star \rightarrow\!\!\!$].

Пусть теперь T будет следующим типом:

$$\begin{aligned} T = & \forall X_0 <: \text{Top} . \forall X_1 <: \text{Top} . \forall X_2 <: \text{Top} . \\ & \neg(\forall Y_0 <: X_0 . \forall Y_1 <: X_1 . \forall Y_2 <: X_2 . \neg X_0) \end{aligned}$$

Рассмотрим утверждение о подтиповании

$$\begin{aligned} \vdash T <: & \forall X_0 <: T . \forall X_1 <: P . \forall X_2 <: Q . \\ & \neg(\forall Y_0 <: \text{Top} . \forall Y_1 <: \text{Top} . \forall Y_2 <: \text{Top} . \\ & \quad \neg(\forall Z_0 <: Y_0 . \forall Z_1 <: Y_1 . \forall Z_2 <: Y_2 . U)) \end{aligned}$$

Это утверждение можно рассматривать как описание состояния некоторого простого компьютера. Переменные X_1 и X_2 являются «регистрами» этой машины. Текущим состоянием этих регистров служат типы P и Q . «Поток команд» машины содержится в третьей строке: первая команда закодирована в границах (Y_2 и Y_1 — обратите внимание на порядок) для переменных Z_1 и Z_2 , а непроясненный тип U представляет оставшиеся команды в программе. Тип T , вложенные отрицания и границы переменных X_0 и Y_0 играют приблизительно ту же роль, что и в более простом предыдущем примере: они позволяют нам «поворнуть рычаг» и вернуться к подцели, имеющей ту же форму, что и исходная цель. Один поворот рычага будет соответствовать одному такту нашей машины.

В этом примере команда в начале потока команд кодирует собой инструкцию «обменять содержимое регистров 1 и 2». Чтобы убедиться в этом, мы с помощью двух установленных нами ранее утверждений проводим следующее вычисление. (Переменные P и Q , содержимое регистров, выделены, чтобы за ними было проще следить.)

$\vdash T$	$\vdash \forall X_0 <: T. \forall X_1 <: P. \forall X_2 <: Q.$
	$\neg(\forall Y_0 <: \text{Top}. \forall Y_1 <: \text{Top}. \forall Y_2 <: \text{Top}.$
	$\neg(\forall Z_0 <: Y_0. \forall Z_1 <: Y_2. \forall Z_2 <: Y_1.U))$
титтк* \vdash	$\neg(\forall Y_0 <: T. \forall Y_1 <: P. \forall Y_2 <: Q. \neg T)$
	$\neg(\forall Y_0 <: \text{Top}. \forall Y_1 <: \text{Top}. \forall Y_2 <: \text{Top}.$ по утверждению 28.5.4
	$\neg(\forall Z_0 <: Y_0. \forall Z_1 <: Y_2. \forall Z_2 <: Y_1.U))$
титтк \vdash	$(\forall Y_0 <: \text{Top}. \forall Y_1 <: \text{Top}. \forall Y_2 <: \text{Top}.$
	$\neg(\forall Z_0 <: Y_0. \forall Z_1 <: Y_2. \forall Z_2 <: Y_1.U))$
	$\vdash (\forall Y_0 <: T. \forall Y_1 <: P. \forall Y_2 <: Q. \neg T)$ по утверждению 28.5.2
титтк \vdash	$\neg(\forall Z_0 <: T. \forall Z_1 <: Q. \forall Z_2 <: P.U)$
	$\vdash \neg T$ по утверждению 28.5.4
титтк $\vdash T$	
	$\vdash (\forall Z_0 <: T. \forall Z_1 <: Q. \forall Z_2 <: P.U)$ по утверждению 28.5.2

Заметим, что в конце этого вывода не только поменялись местами значения P и Q , но в процессе работы команда, которая вызвала этот обмен, была «истраченена», так что в начале потока «подлежащих исполнению» команд оказался тип U . Если теперь в качестве значения U мы выберем тип, имеющий такой же вид, как только что выполненная нами команда,

$$U = \neg(\forall Y_0 <: \text{Top}. \forall Y_1 <: \text{Top}. \forall Y_2 <: \text{Top}.$$

$$\quad \neg(\forall Z_0 <: Y_0. \forall Z_1 <: Y_2. \forall Z_2 <: Y_1.U'))$$

то мы проведем еще один обмен и вернем регистры к их исходному состоянию, прежде чем выполнить U' . Или же мы можем выбрать другое значение U , вызывающее какое-либо другое поведение. Например, если

$$U = \neg(\forall Y_0 <: \text{Top}. \forall Y_1 <: \text{Top}. \forall Y_2 <: \text{Top}.$$

$$\quad \neg(\forall Z_0 <: Y_0. \forall Z_1 <: Y_1. \forall Z_2 <: Y_2.Y_1))$$

то на следующем такте исполнения машины текущее значение регистра 1, т. е. Q , окажется в позиции U — в сущности, будет произведен «косвенный переход» через регистр 1 к потоку команд, который представлен как Q . Обобщив этот прием, можно закодировать условные конструкции и арифметику (операции последователя, предшественника и проверку на ноль).

Собрав все это вместе, мы получаем доказательство неразрешимости путем сведения двухрегистровых машин — простого варианта обыкновенных машин Тьюринга, в которых есть конечное устройство управления и два счетчика, каждый из которых содержит натуральное число, — к утверждениям о подтиповании.

* «Тогда и только тогда, когда...» — прим. перев.

Теорема 28.5.5 [(Pierce, 1994)]: Для каждой двухрегистровой машины M существует такое утверждение о подтиповании $S(M)$, что $S(M)$ выводимо в полной $F_{<:}$, тогда и только тогда, когда вычисление M завершается.

Таким образом, если бы мы могли решить, доказуемо ли произвольное утверждение о подтиповании, мы могли бы также решить, останавливается ли произвольная двухрегистровая машина. Поскольку проблема останова для двухрегистровых машин неразрешима (см. Hopcroft and Ullman, 1979), неразрешима и задача определения подтипования для полной $F_{<:}$.

Следует еще раз подчеркнуть, что неразрешимость отношения подтипования не означает, что полуалгоритм для подтипования, разработанный в §28.4, некорректен либо неполон. Если утверждение $\Gamma \vdash S <: T$ доказуемо согласно декларативным правилам для подтипования, то алгоритм определенно завершится и вернет значение *истина*. Если $\Gamma \vdash S <: T$ не выводится из декларативных правил, то алгоритм либо не завершится, либо вернет значение *ложь*. Каждое данное утверждение о подтиповании может так или иначе оказаться невыводимым: либо оно порождает бесконечную последовательность подцелей (что означает отсутствие конечного вывода с данным заключением), либо сводится к очевидному противоречию вроде $\text{Top} <: S \rightarrow T$. Алгоритм проверки подтипования может распознать один из этих случаев, но не другой.

Означает ли неразрешимость полной $F_{<:}$, что система практически бесполезна? Напротив, как правило, считают, что *сама по себе* неразрешимость $F_{<:}$ не является таким уж серьезным недостатком. Во-первых, было показано (Ghelli, 1995), что, чтобы заставить процедуру проверки подтипования зациклиться, нужно дать ей цель с тремя достаточно экзотическими свойствами, причем трудно представить, что программист случайно напишет хоть какую-то из них. Кроме того, существует немало популярных языков, для которых задача проверки типов в принципе либо чрезвычайно трудоемка — как в ML или Haskell (§22.7), — либо вообще неразрешима, как для C++ или λProlog (Felty, Gunter, Hannan, Miller, Nadathur, and Scedrov, 1988). На практике оказывается, что отсутствие объединений и пересечений, о котором упоминается в следующем разделе (см. упражнение 28.6.3) является намного более серьезным недостатком полной $F_{<:}$, чем неразрешимость.

Упражнение 28.5.6 [★ ★ ★★]: (1) Определите вариант полной $F_{<:}$ без типа Top , но со связываниями типов вида $X <: T$ и вида X (т. е. как с ограниченной, так и с неограниченной квантификацией); этот вариант называется полностью ограниченной квантификацией (*completely bounded quantification*). (2) Покажите, что отношение подтипования для этой системы разрешимо. (3) Дает ли такое ограничение удовлетворительное решение для проблем, обсуждаемых в этом разделе? В частности, будет ли оно работать в языках с дополнительными конструкторами типов, такими как числа, записи, варианты и т. п.?

$\rightarrow \forall < : \text{Top}$	
$\Gamma \vdash S \vee T =$ T если $\Gamma \vdash S <: T$ S если $\Gamma \vdash T <: S$ J если $S = X$ $X <: U \in \Gamma$ $\Gamma \vdash U \vee T = J$ J если $T = X$ $X <: U \in \Gamma$ $\Gamma \vdash S \vee U = J$ $M_1 \rightarrow J_2$ если $S = S_1 \rightarrow S_2$ $T = T_1 \rightarrow T_2$ $\Gamma \vdash S_1 \wedge T_1 = M_1$ $\Gamma \vdash S_2 \vee T_2 = J_2$ $\forall X <: U_1 . J_2$ если $S = \forall X <: U_1 . S_2$ $T = \forall X <: U_1 . T_2$ $\Gamma, X <: U_1 \vdash S_2 \vee T_2 = J_2$ Top в остальных случаях	$\Gamma \vdash S \wedge T =$ S если $\Gamma \vdash S <: T$ T если $\Gamma \vdash T <: S$ $J_1 \rightarrow M_2$ если $S = S_1$ $T = T_1$ $\Gamma \vdash S_1$ $\Gamma \vdash S_2$ $\forall X <: U_1 . M_2$ если $S = \forall X <: U_1 . T_2$ $\Gamma, X <: U_1 \vdash S_2 \vee T_2 = J_2$ <i>неудача</i> в остальных

Рис. 28.5. Алгоритмы поиска объединений и пересечений для ядерной $F_{<}$.

28.6. Объединения и пересечения

В §16.3 мы убедились, что в языках с подтипами желательно существование т.н. *объединения* (join) для всякой пары типов S и T — то есть, типа J , наименьшего среди всех общих надтипов S и T . В этом разделе мы покажем, что отношение подтиповования для ядерной $F_{<}$, действительно имеет объединение для любой пары типов S и T , а также пересечение для любых S и T , имеющих хотя бы один общий подтип, и дадим алгоритмы для их вычисления. (Напротив, оба эти свойства отсутствуют в полной $F_{<:}$; см. упражнение 28.6.3.)

Мы пользуемся записью $\Gamma \vdash S \vee T = J$, означающей « J является объединением типов S и T в контексте Γ », а также $\Gamma \vdash S \wedge T = M$, означающей « M является пересечением S и T в Γ ». Оба алгоритма для вычисления этих отношений определяются на рис. 28.5. Обратите внимание, что некоторые варианты в этих определениях пересекаются; чтобы определения работали как детерминистские алгоритмы, мы объявляем, что всегда выбирается первый подходящий вариант.

Несложно убедиться, что \vee и \wedge являются всюду определенными функциями в том смысле, что \vee всегда возвращает тип, а \wedge всегда либо возвращает тип, либо терпит неудачу. Для этого достаточно заметить, что общий вес (см. определение 28.3.4) типов S и T относительно Γ при рекурсивных вызовах всегда уменьшается.

Теперь следует доказать, что по этим алгоритмам действительно вычисляются объединения и пересечения. Доказательство разбито на две части: утверждение 28.6.1 показывает, что вычисленное объединение является верхней гранью S и T , а пересечение (когда оно есть) является нижней гранью.

Затем утверждение 28.6.2 показывает, что вычисленное объединение меньше любой верхней грани S и T , а пересечение больше любой общей нижней грани (и существует всегда, когда у S и T имеется общая нижняя грань).

Утверждение 28.6.1

1. Если $\Gamma \vdash S \vee T = J$, то $\Gamma \vdash S <: J \wedge \Gamma \vdash T <: J$.
2. Если $\Gamma \vdash S \wedge T = M$, то $\Gamma \vdash M <: S \wedge \Gamma \vdash M <: T$.

Доказательство: Прямолинейная индукция по размеру вывода $\Gamma \vdash S \vee T = J$ или $\Gamma \vdash S \wedge T = M$ (т. е. по количеству рекурсивных вызовов, требуемых для вычисления J или M).

Утверждение 28.6.2

1. Если $\Gamma \vdash S <: V \wedge \Gamma \vdash T <: V$, то $\Gamma \vdash S \vee T = J$ для некоторого J , причем $\Gamma \vdash J <: V$.
2. Если $\Gamma \vdash L <: S \wedge \Gamma \vdash L <: T$, то $\Gamma \vdash S \wedge T = M$ для некоторого M , причем $\Gamma \vdash L <: M$.

Доказательство: Проще всего доказать обе части утверждения одновременной индукцией по размерам алгоритмических выводов утверждений $\Gamma \vdash S <: V \wedge \Gamma \vdash T <: V$ для части 1, и $\Gamma \vdash L <: S \wedge \Gamma \vdash L <: T$ для части 2. (Благодаря теореме 28.3.3 мы можем быть уверены, что алгоритмические соответствия декларативным выводам всегда существуют.)

1. Если какой-либо из двух выводов является экземпляром правила SA-TOP, то $V = \text{Top}$, и требуемый результат, $\Gamma \vdash J <: V$, следует непосредственно.

Если вывод $\Gamma \vdash T <: V$ является экземпляром правила SA-REFL-TVAR, то $T = V$. Но в таком случае первый данный вывод дает нам $\Gamma \vdash S <: V = T$, так что применим первый вариант в определении объединения, и он дает нам $\Gamma \vdash S \vee T = T$, что удовлетворяет требованиям. Аналогично, если вывод $\Gamma \vdash S <: V$ является экземпляром SA-REFL-TVAR, то $S = V$. Но тогда второй данный вывод говорит, что $\Gamma \vdash T <: V = S$, так что применим второй вариант в определении объединения, и он дает нам $\Gamma \vdash S \vee T = S$, что опять же удовлетворяет требованиям.

Если вывод $\Gamma \vdash S <: V$ заканчивается экземпляром правила SA-TRANS-TVAR, то мы имеем $S = X$, причем $X <: U \in \Gamma$, и имеется подвывод $\Gamma \vdash U \vee T = J$. Третий вариант в определении объединения дает $\Gamma \vdash S \vee T = J$, а из индуктивного предположения мы имеем $\Gamma \vdash J <: V$. Аналогично мы рассуждаем и в том случае, когда $\Gamma \vdash T <: V$ заканчивается на SA-TRANS-TVAR.

Теперь по форме алгоритмических правил подтипования несложно видеть, что остаются только варианты, в которых оба данных вывода завершаются либо правилом SA-ARROW, либо SA-ALL.

Если оба вывода завершаются SA-ARROW, то мы имеем $S = S_1 \rightarrow S_2$, $T = T_1 \rightarrow T_2$ и $V = V_1 \rightarrow V_2$, причем $\Gamma \vdash V_1 <: S_1$, $\Gamma \vdash S_2 <: V_2$, $\Gamma \vdash V_1 <: T_1$ и $\Gamma \vdash T_2 <: V_2$. Согласно части (2) предположения индукции, $\Gamma \vdash S_1 \wedge T_1 = M_1$ для некоторого M_1 , причем $\Gamma \vdash V_1 <: M_1$, а согласно части (1), $\Gamma \vdash S_2 \vee T_2 = J_2$ для некоторого J_2 , причем $\Gamma \vdash J_2 <: M_2$. Пятый вариант в определении объединений дает $\Gamma \vdash S_1 \rightarrow S_2 \vee T_1 \rightarrow T_2 = M_1 \rightarrow J_2$, а по правилу S-ARROW мы имеем $\Gamma \vdash M_1 \rightarrow J_2 <: V_1 \rightarrow V_2$.

Наконец, если оба данных вывода заканчиваются на SA-ALL, то мы имеем $S = \forall X <: U_1 . S_2$, $T = \forall X <: U_1 \rightarrow T_2$ и $V = \forall X <: U_1 . V_2$, причем $\Gamma, X <: U_1 \vdash S_2 <: V_2$ и $\Gamma, X <: U_1 \vdash T_2 <: V_2$. Согласно части (1) предположения индукции, $\Gamma, X <: U_1 \vdash S_2 \vee T_2 = J_2$, причем $\Gamma, X <: U_1 \vdash J_2 <: V_2$. Шестой вариант в определении объединений дает нам $J = \forall X <: U_1 . J_2$, а по правилу S-ALL имеем $\Gamma \vdash \forall X <: U_1 . J_2 <: \forall X <: U_1 . V_2$.

2. Если вывод $\Gamma \vdash L <: T$ заканчивается на SA-TOP, то тип T равен Top , так что $\Gamma \vdash S <: T$, и, по первому варианту в определении пересечения, $\Gamma \vdash S \wedge T = S$. Однако из второго данного вывода мы знаем, что $\Gamma \vdash L <: S$, так что требуемое утверждение доказано. Аналогично рассуждаем и в случае, когда вывод $\Gamma \vdash L <: S$ заканчивается на SA-TOP.

Если вывод $\Gamma \vdash L <: S$ заканчивается на SA-REFL-TVAR, то $L = S$, и второй данный вывод дает $\Gamma \vdash L = S <: T$, откуда по определению пересечения мы имеем $\Gamma \vdash S \wedge T = S$, так что доказательство закончено. Аналогично рассуждаем, когда вывод $\Gamma \vdash L <: T$ заканчивается на SA-REFL-TVAR.

Остаются только варианты, в которых оба данных вывода заканчиваются на SA-TRANS-TVAR, SA-ARROW или SA-ALL.

Если оба вывода заканчиваются на SA-TRANS-TVAR, то имеем $L = X$, причем $X <: U \in \Gamma$, и имеются два подвывода $\Gamma \vdash U <: S$ и $\Gamma \vdash U <: T$. Согласно части (2) предположения индукции, $\Gamma \vdash U <: M$, откуда имеем $\Gamma \vdash L <: M$ по правилу S-TVAR и транзитивности.

Если оба вывода заканчиваются на SA-ARROW, то мы имеем $S = S_1 \rightarrow S_2$, $T = T_1 \rightarrow T_2$ и $L = L_1 \rightarrow L_2$, причем $\Gamma \vdash S_1 <: L_1$, $\Gamma \vdash L_2 <: S_2$, $\Gamma \vdash T_1 <: L_1$ и $\Gamma \vdash L_2 <: T_2$. Согласно части (1) предположения индукции, $\Gamma \vdash S_1 \vee T_1 = J_1$ для некоторого J_1 , причем $\Gamma \vdash J_1 <: L_1$, а согласно части (2), $\Gamma \vdash S_2 \wedge T_2 = M_2$, причем $\Gamma \vdash L_2 <: M_2$. Определение пересечений дает нам $\Gamma \vdash S_1 \rightarrow S_2 \wedge T_1 \rightarrow T_2 = J_1 \rightarrow M_2$, а по правилу S-ARROW, $\Gamma \vdash L_1 \rightarrow L_2 <: J_1 \rightarrow M_2$.

Аналогично ведется доказательство и в варианте с правилом SA-ALL.

Упражнение 28.6.3 [РЕКОМЕНДУЕТСЯ, ★★★]: Рассмотрим пару типов, предложенных Гелли (*Ghelli, 1990*): $S = \forall X <: Y \rightarrow Z . Y \rightarrow Z$ и $T = \forall X <: Y' \rightarrow Z' . Y' \rightarrow Z'$ и контекст $\Gamma = Y <: \text{Top}, Z <: \text{Top}, Y' <: Y, Z' <: Z$. (1) Сколько в полной $F_{<}$ существует типов, являющихся подтипами как S , так и T в контексте Γ ?

(2) Покажите, что в полной F_{\leq} , типы S и T не имеют пересечения в контексте Γ . (3) Найдите пару типов, которые в полной F_{\leq} не имеют объединения в контексте Γ .

28.7. Ограничные кванторы существования

Чтобы расширить алгоритм ядерной F_{\leq} , на язык с кванторами существования, нужно преодолеть одну дополнительную трудность. Напомним декларативное правило устранения для экзистенциальных типов:

$$\frac{\Gamma \vdash t_1 : \{\exists X : T_{11}, T_{12}\} \quad \Gamma, X : T_{11}, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2} \quad (\text{T-UPACK})$$

В §24.1 мы отметили, что типовая переменная X присутствует в контексте, в котором во второй предпосылке вычисляется тип терма t_2 , но *отсутствует* в контексте заключения правила. Это означает, что тип T_2 не должен содержать X в качестве свободной переменной, поскольку всякое такое вхождение в заключении окажется вне области видимости. Более подробно этот вопрос обсуждался в §25.5, где мы указали, что изменение контекста от предпосылки к заключению соответствует *отрицательному сдвигу* индексов переменных в T_2 , если мы представим термы в безымянном формате де Брауна; этот сдвиг будет неудачен, если X содержится в T_2 как свободная переменная.

Как это обстоятельство влияет на алгоритм минимальной типизации для языка с экзистенциальными типами? В частности, что нам делать с таким выражением, как $t = \text{let } \{X, x\} = p \text{ in } x$, в котором p имеет тип $\{\exists X, \text{Nat} \rightarrow X\}$? Наиболее естественным типом тела x будет $\text{Nat} \rightarrow X$, упоминающий связанный переменной X . Однако согласно декларативному отношению типизации (с правилом включения), x имеет также типы $\text{Nat} \rightarrow \text{Top}$ и Top . Поскольку ни в одном из этих типов X не встречается, всему терму t можно в декларативной системе присвоить типы $\text{Nat} \rightarrow \text{Top}$ и Top . В общем случае, в выражении распаковки всегда можно расширить тело до типа, в котором не содержится свободной переменной X , а затем применить T-UPACK. Так что, если мы хотим, чтобы наш алгоритм минимальной типизации был полон, он должен не просто объявлять об ошибке при работе с выражением распаковки, где наименьший тип тела T_2 содержит свободное вхождение связанной переменной X . Вместо этого он должен попытаться расширить T_2 до некоторого типа, в котором X не упоминается. Ключевое наблюдение, необходимое для того, чтобы такая идея сработала, состоит в том, что множество надтипов данного типа, не содержащих X , всегда имеет минимальный элемент. Это показывает следующее упражнение (решение которого было найдено Гелли и Пирсом, Ghelli and Pierce, 1998).

Упражнение 28.7.1 [★★★]: Постройте алгоритм, вычисляющий в ядерной F_{\leq} , с ограниченными экзистенциальными типами наименьший надтип данного типа T , не содержащий переменной X , по отношению к данному контексту Γ . Такой надтип обозначается $R_{X, \Gamma}(T)$.

Теперь алгоритмическое правило типизации для устранения кванторов существования можно записать так:

$$\frac{\Gamma \mapsto t_1 : T_1 \quad \Gamma \mapsto T_1 \uparrow \{\exists X <: T_{11}, T_{12}\} \\ \Gamma, X <: T_{11}, x: T_{12} \mapsto t_2 : T_2 \quad R_{X, (\Gamma, X <: T_{11}, x: T_{12})}(T_2) = T'_2}{\Gamma \mapsto \text{let } \{X, x\} = t_1 \text{ in } t_2 : T'_2} \quad (\text{TA-UNPACK})$$

Как и следовало ожидать, в *полной* $F_{<:}$ с ограниченными кванторами существования ситуация сложнее. Гелли и Пирс (Ghelli and Pierce, 1998) приводят пример типа T , контекста Γ и переменной X , таких, что множество надтипов T , не содержащих X , не имеет минимального элемента в контексте Γ . Отсюда немедленно следует, что отношение типизации в такой системе не обладает наименьшими типами.

Упражнение 28.7.2 [★★★]. Покажите, что отношение подтиpирования для варианта полной $F_{<:}$, имеющего только кванторы существования (т. е. без кванторов общности) также неразрешимо.

28.8. Ограниченнaя квантификация и тип Bot

Добавление наименьшего типа Bot несколько усложняет метатеоретические свойства $F_{<:}$. Это происходит оттого, что в типе вида $\forall X <: \text{Bot}. T$ внутри T переменная X является, на самом деле, *синонимом* Bot , поскольку по предположению X — подтип Bot , а Bot , по правилу S-Bot, — подтип X . Это, в свою очередь, означает, что пары типов вроде $\forall X <: \text{Bot}. X \rightarrow X$ и $\forall X <: \text{Bot}. \text{Bot} \rightarrow \text{Bot}$ эквивалентны в отношении подтиpирования, хотя синтаксически они различаются. Более того, если окружающий контекст содержит предположения $X <: \text{Bot}$ и $Y <: \text{Bot}$, то типы $X \rightarrow Y$ и $Y \rightarrow X$ эквивалентны, хотя ни в одном из них Bot явно не упоминается. Несмотря на все это, даже при наличии Bot все основные свойства $F_{<:}$ все равно сохраняются. Детали можно найти в Pierce, 1997a.

Часть VI

Системы высших порядков

Глава 29

Операторы над типами и виды

В предыдущих главах мы, чтобы облегчить чтение примеров, часто употребляли сокращения вроде

```
CBool = ∀X. X → X → X
```

и

```
Pair Y Z = ∀X. (Y → Z → X) → X
```

и писали, скажем, $\lambda x: \text{Pair } \text{Nat} \text{ Bool}. x$, вместо более громоздкого $\lambda x: (\text{Nat} \rightarrow \text{Bool} \rightarrow X) \rightarrow X. x$.

Запись `CBool` — просто сокращение; когда она встречается в примере, следует просто заменить её на правую часть определения. Напротив, запись `Pair` представляет собой *параметризованное сокращение* (parametric abbreviation); когда мы видим `Pair S T`, требуется в его определении заменить параметры `Y` и `Z` на реальные типы `S` и `T`. Другими словами, параметрические сокращения вроде `Pair` дают нам неформальный способ определения функций на уровне выражений типа.

Мы уже использовали выражения на уровне типа, например `Array T` или `nbRef T`. В них используются *конструкторы типов* (type constructors) `Array` и `Ref`. Несмотря на то, что эти конструкторы не определены программистом, а встроены в язык, они также являются разновидностью функций на уровне типов. Можно рассматривать `Ref` как функцию, которая для каждого типа `T` выдает тип ссылочных ячеек, содержащих элементы `T`.

В этой и двух последующих главах нашей задачей будет более формальное рассмотрение таких функций на уровне типа, которые называются *операторами над типами* (type operators). В этой главе мы вводим основные механизмы абстракции и применения на уровне типов; точное определение

В этой главе вводится простое типизированное лямбда-исчисление с операторами типов, $\lambda\omega$ (рис. 29.1). В примерах, кроме того, используются числа и булевские значения (8.2), а также универсальные типы (23.1). Соответствующая реализация на OCaml называется `fullomega`.

того, когда следует считать два выражения типа эквивалентными; а также отношение корректности построения, называемое *отношением видообразования* (kinding), которое не позволяет писать бессмысленные типовые выражения. Глава 30 идет еще на шаг дальше; там операторы над типами рассматриваются как *полноправные элементы языка* (first-class citizens) — т. е. как сущности, которые можно передавать аргументами в функции. В этой главе вводится известная Система F_ω , которая обобщает квантификацию по типам из Системы F (глава 23) до *квантификации высшего порядка* (higher-order quantification) по операторам типов. В главе 31 рассматривается сочетание операторов над типами, квантификации высшего порядка и подтипов.

29.1. Неформальное введение

При изучении функций на уровне типов нам прежде всего потребуется нотация для абстракции и применения типов. Как правило, для этого используется *та же* запись, то и для абстракции и применения на уровне термов: абстракция обозначается через λ , а применение обозначается написанием термов рядом друг с другом, через пробел.¹ Например, запись $\lambda X. \{a:X, b:X\}$ обозначает функцию, которая принимает тип T и выдает тип записей $\{a:T, b:T\}$. Применение этой функции к типу Bool записывается в виде $(\lambda X. \{a:X, b:X\}) \text{Bool}$.

Подобно обычным функциям, типовые функции от нескольких аргументов получаются из одноаргументных посредством *каррирования* (currying). Например, выражение типа $\lambda Y. \lambda Z. \forall X. (Y \rightarrow Z \rightarrow X) \rightarrow X$ обозначает двухместную функцию — или, строго говоря, одноместную функцию, которая, будучи применена к типу S, дает другую одноместную функцию, которая при применении к типу T дает тип $\forall X. (S \rightarrow T \rightarrow X) \rightarrow X$.

Мы будем продолжать пользоваться неформальными сокращениями для длинных выражений типа, включая операторы над типами. Например, в оставшейся части этой главы мы будем считать, что имеется сокращение

$\text{Pair} = \lambda Y. \lambda Z. \forall X. (Y \rightarrow Z \rightarrow X) \rightarrow X;$

Так что, когда мы в примерах пишем $\text{Pair } S \ T$, имеется в виду

$(\lambda Y. \lambda Z. \forall X. (Y \rightarrow Z \rightarrow X) \rightarrow X) \ S \ T.$

Другими словами, вместо неформального соглашения о параметрических сокращениях, которым мы пользовались до сих пор, теперь у нас будет более простое неформальное соглашение, что все простые сокращения заменяются на свои правые части, а также формальные механизмы для определения

¹ У подобной нотационной экономии есть тот недостаток, что терминология для выражений различного вида иногда оказывается довольно запутанной. Например, выражение «абстракция типа» может теперь обозначать абстракцию, ожидающую тип в качестве аргумента (т. е. терм вида $\lambda X. t$), а может — абстракцию на уровне типов (т. е. выражение типа вроде $\lambda X. \{a:X\}$). В контекстах, где эти варианты можно перепутать, обычно предпочтают использовать в первом смысле выражение «полиморфная функция», а во втором — выражение «абстракция на уровне типов» или «операторная абстракция».

и конкретизации операторов над типами. Операции определения и раскрытия сокращений также можно формально определить — т. е. их можно сделать операциями целевого языка, а не соглашениями метаязыка, — но этим мы здесь заниматься не будем. Заинтересованные читатели могут обратиться к литературе по системам типов с *определениями* (definitions) или *одноэлементными видами* (singleton kinds); см. труды Севери и Полла (Severi and Poll, 1994), Стоуна и Харпера (Stone and Harper, 2000), Крэри (Crary, 2000), а также приведенные в них ссылки.

Абстракция и применение на уровне типов позволяет нам записывать один и тот же тип несколькими различными способами. Например, если Id служит сокращением для оператора над типами $\lambda X.X$, то выражения

$$\begin{array}{lll} \text{Nat} \rightarrow \text{Bool} & \text{Nat} \rightarrow \text{Id Bool} & \text{Id Nat} \rightarrow \text{Id Bool} \\ \text{Id Nat} \rightarrow \text{Bool} & \text{Id } (\text{Nat} \rightarrow \text{Bool}) & \text{Id } (\text{Id } (\text{Id Nat} \rightarrow \text{Bool})) \end{array}$$

являются разными именами одного и того же функционального типа. Чтобы сделать это интуитивное понятие формально точным, мы вводим отношение *эквивалентности определений* (definitional equivalence) для типов; она обозначается как $S \equiv T$. Самое важное уравнение в определении этого отношения

$$(\lambda X : K_{11}. T_{12}) T_2 \equiv [X \mapsto T_2] T_{12} \quad (\text{Q-APPABs})$$

говорит, что абстракция на уровне типов, примененная к аргументу, эквивалентна телу этой абстракции, в которое вместо формального параметра подставлен аргумент. Эквивалентность определений используется при проверке типов в новом правиле

$$\frac{\Gamma \vdash t : S \quad S \equiv T}{\Gamma \vdash t : T} \quad (\text{T-EQ})$$

формально выражают интуитивную идею о том, что если два типа эквивалентны, то элементы одного из них являются также элементами другого.

Еще одна новая возможность, проистекающая из наличия механизмов абстракции и применения типов, — это способность писать бессмысленные типовые выражения. Например, применение одного обычного типа к другому, скажем, в типовом выражении (Bool Nat), столь же абсурдно, как применение `true` к 6 на уровне термов. Чтобы избежать такой бессмыслицы, мы вводим систему *видов* (kinds), согласно которой типы классифицируются по количеству аргументов («по арности»), так же как функциональные типы указывают нам количество аргументов у термов.

Виды строятся на основе одного элементарного вида, который обозначается символом $*$ («звездочка») и читается как «тип», а также одного двуместного конструктора \Rightarrow . Видами являются, например,

- * вид простых типов (скажем, Bool или $\text{Bool} \rightarrow \text{Bool}$)
- $* \Rightarrow *$ вид операторов над типами (т. е. функций, которые переводят простые типы в простые типы)
- $* \Rightarrow * \Rightarrow *$ вид функций, которые переводят простые типы в операторы над типами (т. е. вид двуместных операторов)
- $(* \Rightarrow *) \Rightarrow *$ вид функций, которые переводят операторы над типами в простые типы.

Таким образом, виды являются «типами типов». В сущности, система видов представляет собой копию простого типизированного лямбда-исчисления «уровнем выше».

В дальнейшем мы будем использовать слово *тип* (type) для любого выражения на уровне типов — т. е. как для обычных типов, таких как $\text{Nat} \rightarrow \text{Nat}$ и $\forall X. X \rightarrow X$, так и для операторов над типами, вроде $\lambda X. X$. Когда нам будет нужно говорить об обычных типах (т. е. о выражениях типа, которые используются при классификации термов), мы будем называть их *простыми типами* (proper types).

Выражения типа с такими видами как $(\ast \Rightarrow \ast) \Rightarrow \ast$ называются *операторами высших порядков над типами* (higher-order type operators). В отличие от функций высших порядков на уровне термов, которые часто весьма полезны, операторы высших порядков над типами являются скорее экзотикой. Один класс примеров их использования мы встретим в главе 32.

Чтобы упростить задачу проверки корректности видов для выражений типа, каждую абстракцию на уровне типов мы снабжаем указанием на вид ее связанной переменной. Например, официальное определение оператора *Pair* таково:

```
Pair = λA::*. λB::*. ∀X. (A → B → X) → X;
```

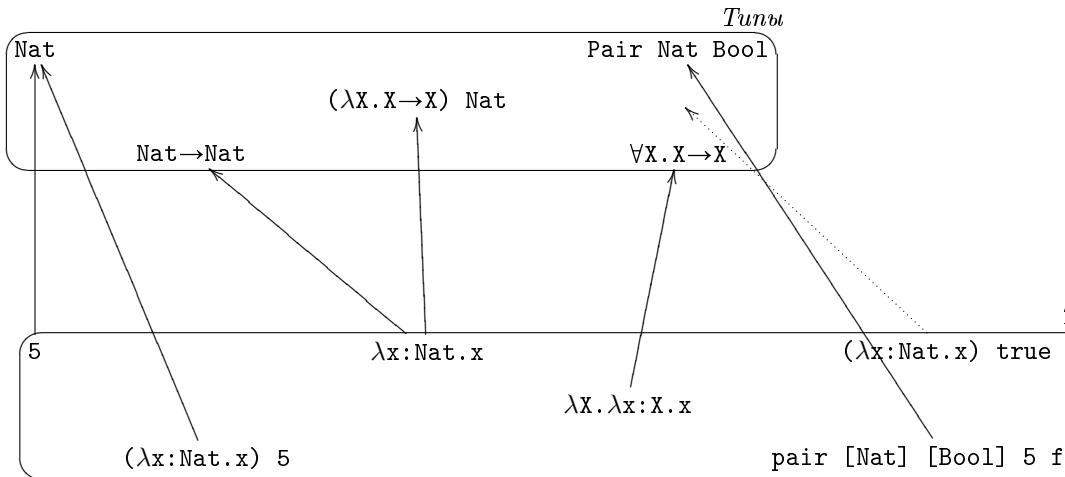
(Обратите внимание на двойное двоеточие.) Однако, поскольку почти все такие аннотации будут иметь вид \ast , мы продолжим писать $\lambda X.$ Т как сокращение для $\lambda X::\ast.$ Т.

Прояснить картину можно при помощи нескольких диаграмм. Выражения нашего языка теперь делятся на три отдельных класса: термы, типы и виды. На уровне термов мы имеем элементарные значения (целые, числа с плавающей точкой и т. п.), составные значения (записи и т. п.), абстракции и применения на уровне значений, абстракции типов и применения типов:

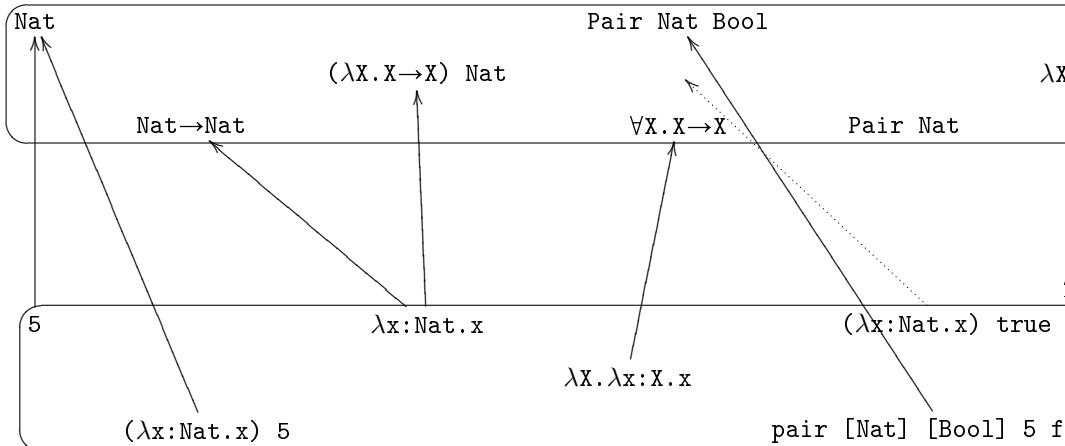
Термы

5	$\lambda x:\text{Nat}. x$	$(\lambda x:\text{Nat}. x) \text{ true}$
	$\lambda X. \lambda x:X. x$	
$(\lambda x:\text{Nat}. x) 5$		$\text{pair } [\text{Nat}] [\text{Bool}] 5 \text{ false}$

На уровне типов имеются две разновидности выражений. Во-первых, есть простые типы вроде Nat , $\text{Nat} \rightarrow \text{Nat}$, Pair Nat Bool и $\forall X. X \rightarrow X$, каждый из которых содержит термы. (Разумеется, не у всех термов имеется тип; например, его нет у $(\lambda x:\text{Nat}. x) \text{ true}$.)



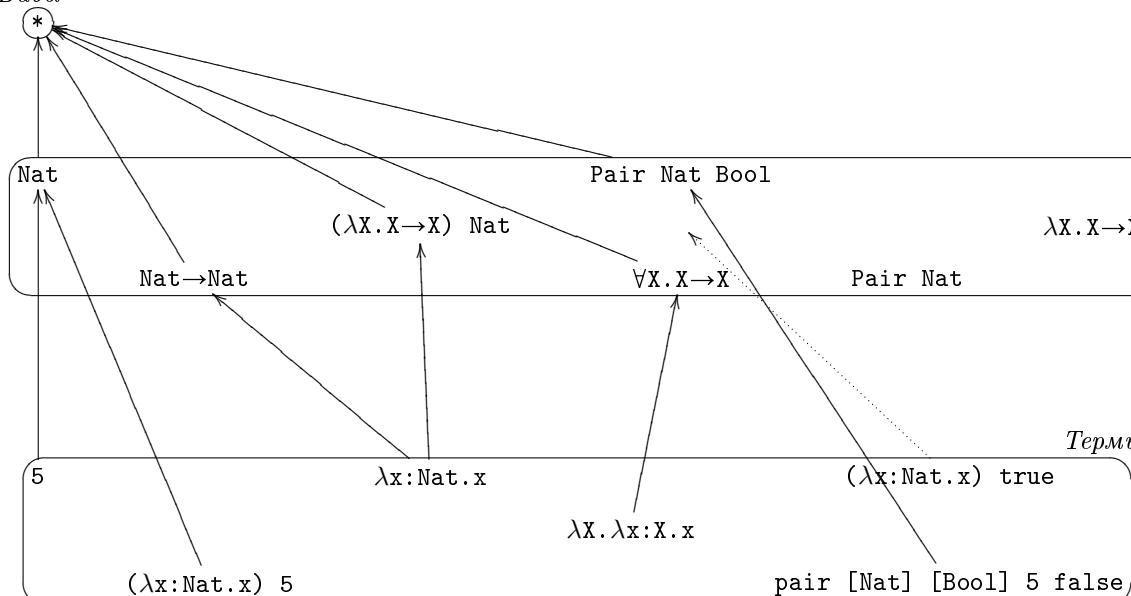
Существуют также операторы над типами, такие как Pair и $\lambda X. X \rightarrow X$, которые сами по себе *не служат* для классификации термов (то есть, не имеет смысла вопрос «Какие термы имеют тип $\lambda X. X \rightarrow X$?»). Однако они могут быть применены к аргументам, давая при этом простые типы, такие как $(\lambda X. X \rightarrow X)$ Nat, которые, в свою очередь, используются при классификации термов.



Заметим, что простые типы, т. е. типовые выражения вида $*$, могут в качестве синтаксических составляющих включать операторы высшего порядка над типами, например, $(\lambda X. X \rightarrow X)$ Nat или Pair Nat Bool. Точно так же, термовые выражения, принадлежащие базовым типам вроде Nat, могут в качестве подвыражений содержать лямбда-абстракции, например, $(\lambda x: \text{Nat}. x)$ 5.

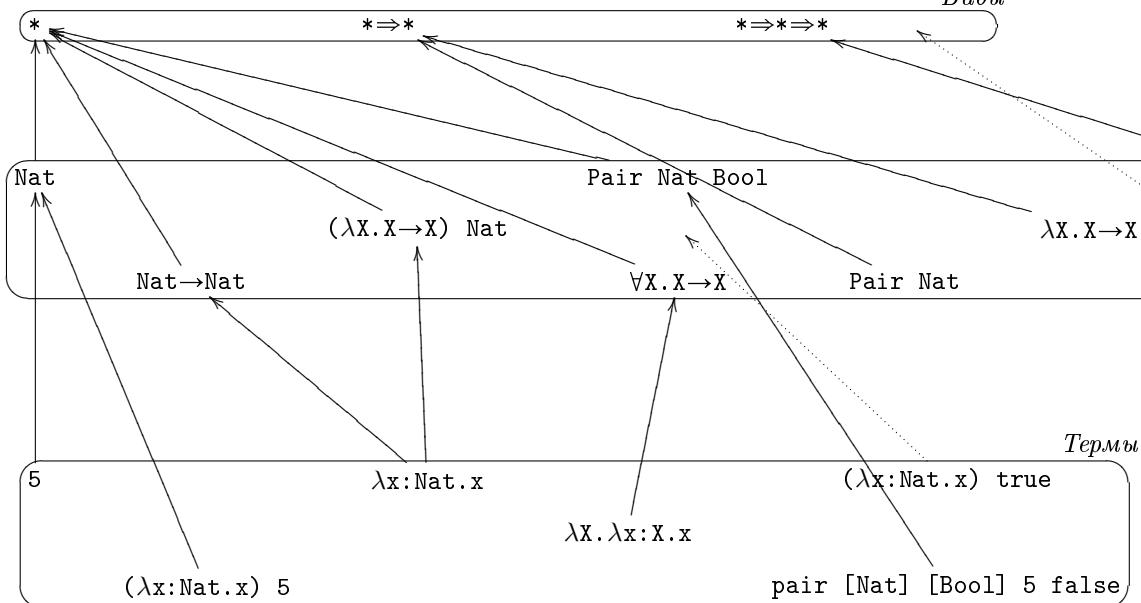
Наконец, имеется уровень видов. Простейшим видом является $*$; он содержит в качестве членов все простые типы.

Виды



Операторы над типами, скажем, $\lambda X. X -> X$ или `Pair`, принадлежат функциональным видам, таким как $* \Rightarrow *$ или $* \Rightarrow * \Rightarrow *$. Неверно построенные выражения на уровне типов, вроде `Pair Pair`, ни к какому виду не принадлежат.

Виды



Упражнение 29.1.1 [★]: Чем отличаются по смыслу такие выражения типового уровня, как $\forall X. X -> X$ и $\lambda X. X -> X$?

Упражнение 29.1.2 [★]: Почему функциональный тип `Nat -> Nat` не принадлежит к функциональному виду вроде $* \Rightarrow *$?

Теперь естественно будет спросить: «Зачем останавливааться на третьем уровне выражений?». Нельзя ли ввести функции, переводящие виды в виды, применение на уровне видов, и т. п., добавить четвертый уровень для классификации видов в соответствии с их функциональным построением, и продолжать в том же духе до бесконечности? Системы такого рода исследовались в рамках теории *систем чистых типов* (pure type systems) (Terlouw, 1989; Berardi, 1988; Barendregt, 1991, 1992; Jutting, McKinna, and Pollack, 1994; McKinna and Pollack, 1993; Pollack, 1994). Однако для языков программирования трех уровней оказалось достаточно.

Действительно, при том, что операторы над типами в той или иной форме присутствуют практически во всех статически типизированных языках программирования, проектировщики языков редко дают программистам доступ ко всем выразительным возможностям даже нашей теперешней системы. В некоторых языках (например, в Java), есть лишь ограниченный набор встроенных операторов вроде `Array`, и нет возможности определять новые. В других языках операторы над типами совмещены с другими языковыми конструкциями; например, в ML такие операторы включены в механизм `datatype`. Разрешается определять *параметрические типы данных* (parametric datatypes), такие как²

```
type 'a Tuor = tyoptag of ('a → 'a);
```

что мы бы записали как

```
Tuor = λX. <tyoptag:X → X>;
```

Другими словами, в ML можно определять параметрические варианты, но не произвольные параметрические типы. Это ограничение имеет то преимущество, что каждый раз, когда оператор `Tuor` встречается на уровне типов, на уровне термов будет присутствовать соответствующий тег `tyoptag`. Таким образом, везде, где компилятор должен с помощью отношения эквивалентности определений свести тип вроде `Tuor Nat` к редуцированной форме `Nat → Nat`, в программе находится метка `tyoptag`. Это существенно упрощает алгоритм проверки типов.³

Здесь мы рассматриваем только один конструктор видов, \Rightarrow . Однако в литературе исследовалось и множество других; спектр систем видеообразования, способных проверять и отслеживать различные свойства типовых выражений, сравним со спектром систем типов, проверяющих и отслеживающих свойства термов. Имеются *виды записей* (record kinds) (их элементами являются записи типов, которые не следует путать с типами записей — через такие

² В этом примере мы игнорируем соглашения ML по использованию больших и малых букв в идентификаторах. На самом деле, в OCaml это определение записывалось бы так:

```
type 'a tuor = Tuortag of ('a → 'a);
```

³ Это ограничение аналогично тому, как в ML обрабатываются рекурсивные типы (см. §20.1). Объединение рекурсивных типов с определениями `datatype` дает программисту удобство эквирекурсивных типов, а компилятору — простоту типов изорекурсивных, поскольку аннотации `fold/unfold` упрыгиваются в операции пометки и операторы разбора случаев, необходимые при работе с вариантными типами.

виды естественным образом определяются системы взаимно рекурсивных типов); *строчные виды* (row kinds) (которые описывают «строки полей» — с их помощью можно сконструировать типы записей в системах, поддерживающих полиморфизм через строчные переменные, см. с. 367); *степенные виды* (power kinds), они же *степенные типы* (power types) (через них строится альтернативный подход к подтипам, см. Cardelli, 1988a); *одноэлементные виды* (singleton kinds) (связанные с *определениями* (definitions) — см. с. 471, а также с системами модулей с *совместным использованием* (sharing) — см. с. 493); *зависимые виды* (dependent kinds) («высокоуровневый» аналог зависимых типов, описываемых в §30.5); и многие другие.

29.2. Определения

Полное определение базового лямбда-исчисления с операторами над типами приведено на рис. 29.1. На уровне термов это исчисление содержит только переменные, абстракцию и применение из простого типизированного лямбда-исчисления (и поэтому называется *простым типизированным лямбда-исчислением с операторами над типами*, simply typed lambda-calculus with type operators). На уровне типов мы имеем обычные функциональные типы и типовые переменные, а также абстракцию и применение операторов. Кванторные типы вроде $\forall X.T$ в эту систему не включены; мы подробно рассмотрим их в главе 30.

Наше представление системы с видами расширяет конструкцию простого типизированного лямбда-исчисления в трех отношениях. Во-первых, вводится набор правил *присвоения видов* (kinding), указывающих, как можно сочетать выражения типа и получать при этом новые выражения типа. Запись $\Gamma \vdash T :: K$ означает «тип T принадлежит виду K в контексте Γ ». Обратите внимание на сходство этих правил с правилами типизации для исходного простого типизированного лямбда-исчисления (рис. 9.1).

Во-вторых, каждый раз, когда внутри терма встречается тип T (в форме $\lambda x:T.t$), требуется проверить, правильно ли T образован. Поэтому к старому правилу $T\text{-Abs}$ добавляется новая предпосылка, $\Gamma \vdash t :: *$. Заметим, что T должен иметь в точности вид $*$, т. е. должен быть простым типом, поскольку используется для характеристики значений, которые разрешено принимать переменной x . Правила типизации соблюдают инвариант, согласно которому всякий раз, когда мы можем вывести утверждение $\Gamma \vdash t :: T$, выводимо будет и утверждение $\Gamma \vdash T :: *$ (при условии, что все упоминаемые в контексте типы имеют правильные виды). Этот момент подробнее обсуждается в §30.3.

В-третьих, добавляется набор правил, касающихся отношения *эквивалентности определений* (definitional equivalence) между типами. Мы пишем $S \equiv T$, что читается «*определения типов S и T эквивалентны*». Это отношение очень похоже на отношение редукции на уровне термов. Влияние эквивалентности определений на типизацию отражено в новом правиле $T\text{-Eq}$. Предпосылка о видах (опущенная в предыдущем разделе, где мы обсуждали это правило) поддерживает вышеупомянутый инвариант — «термы с правильными типами всегда имеют типы с правильными видами». Заметим, что это правило

похоже на правило включения (T-SUB) в системах с подтипами.

Для доказательства основных метатеоретических свойств этой системы требуется некоторая работа, поскольку отношение эквивалентности между типами дает значительную гибкость в «формах» типов, присваиваемых термам. Мы откладываем исследование этой теории до главы 30.

$\rightarrow \Rightarrow$ Расширяет λ_\rightarrow (9.1)

Синтаксис

$t ::=$	термы:
x	переменная
$\lambda x:T.t$	абстракция
$t t$	применение

$v ::=$	значения:
$\lambda x:T.t$	значение-абстракция

$T ::=$	типы:
X	типовая переменная
$\lambda X::K.T$	абстракция
<i>оператора</i>	
$T T$	применение оператора
$T \rightarrow T$	тип функций

$\Gamma ::=$	контексты:
\emptyset	пустой контекст
$\Gamma, x:T$	связывание термовой переменной
$\Gamma, [X::K]$	связывание типовой переменной

$K ::=$	виды:
$*$	вид простых типов
$K \Rightarrow K$	вид операторов

Вычисление

$$t \rightarrow t'$$

$$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2} \quad (\text{E-APP2})$$

$(\lambda x:T_{11}.t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12}$	(E-APP ABS)
	$\boxed{\Gamma \vdash t : T}$

$x : T \in \Gamma$	(T-VAR)
$\boxed{\Gamma \vdash x : T}$	

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x:T_1 \vdash t_2:T_2}{\Gamma \vdash \lambda x:T_1.t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

Rev: -revision-, November 28, 2010

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad (\text{T-EQ})$$

Присвоение видов $\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \quad (\text{K-TVAR})$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1.T_2 :: K_1 \Rightarrow K_2} \quad (\text{K-ABS})$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \quad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1 \ T_2 :: K_{12}} \quad (\text{K-APP})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *} \quad (\text{K-ARROW})$$

Эквивалентность типов $\boxed{S \equiv T}$

$$T \equiv T \quad (\text{Q-REFL})$$

$$\frac{T \equiv S}{S \equiv T} \quad (\text{Q-SYMM})$$

$$\frac{S \equiv U \quad U \equiv T}{S \equiv T} \quad (\text{Q-TRANS})$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2} \quad (\text{Q-ARROW})$$

$$\frac{S_2 \equiv T_2}{\lambda X :: K_1.S_2 \equiv \lambda X :: K_1.T_2} \quad (\text{Q-ABS})$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2} \quad (\text{Q-APP})$$

 $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}$

Глава 30

Полиморфизм высших порядков

После того, как в главе 29 мы добавили к λ_{\rightarrow} операторы над типами, на следующем шаге было бы естественно сочетать их с другими типовыми конструкциями, изученными нами на протяжении этой книги. В этой главе мы добавляем операторы над типами к полиморфизму Системы F и получаем широко известную систему, которая называется F_{ω} (Girard, 1972). В главе 31 исчисление обогащается подтипами, и получается Система F_{ζ}^{ω} . Эта последняя служит основой нашего последнего расширенного примера: чисто функциональных объектов в главе 32.

Определение F_{ω} строится простой комбинацией возможностей λ_{ω} и Системы F. Однако доказательство основных свойств этой системы (в частности, сохранения и продвижения) требует больших усилий, чем в большинстве виденных нами систем, поскольку приходится учитывать, что проверка типов теперь требует вычислений на уровне типов. Построение этих доказательств и является основной задачей данной главы.

30.1. Определения

Система F_{ω} получается при сочетании Системы F из главы 23 и λ_{ω} из главы 29, с добавлением аннотаций вида $(X:: K)$ всюду, где связываются типовые переменные (т. е. в абстракциях типа и кванторах). Формальное определение системы, содержащей только кванторы всеобщности (но не существования), дано на рис. 30.1. Мы приводим все правила полностью, чтобы облегчить чтение доказательств в §30.3, несмотря на то, что отличия от предыдущих систем минимальны.

Мы используем сокращенную запись $\forall X.T$ для $\forall X:: *.T$, а $\{ \exists X, T \}$ — для $\{ \exists X:: *, T \}$, чтобы термы Системы F можно было напрямую рассматривать в

Примеры из этой главы являются термами F_{ω} (рис. 30.1) с записями, булевскими значениями и кванторами существования (30.2). Соответствующая реализация на OCaml называется *fullomega*. Для зависимых типов, упоминаемых в §30.5, реализация отсутствует.

качестве термов F_ω .

Аналогично можно получить экзистенциальные типы высших порядков путем обобщения X до $X::K$ в исходном представлении кванторов существования из главы 24. Это расширение приведено на рис. 30.2.

30.2. Пример

Расширенный пример программирования с использованием абстракций, включающих операторы над типами, мы увидим в главе 32. Здесь мы рассмотрим намного более скромный пример.

Вспомним кодирование абстрактных типов данных при помощи экзистенциальных типов из §24.2. Теперь допустим, что мы хотим реализовать АТД пар, точно так же, как раньше мы реализовывали АТД таких типов, как счетчики. Новый АТД должен предоставлять операции для построения пар и для доступа к их элементам. Более того, хотелось бы, чтобы эти операции были *полиморфны*, чтобы с их помощью можно было создавать и использовать пары элементов любых типов S и T . Таким образом, предоставляемый нами абстрактный тип должен быть не простым типом, а абстрактным *конструктором типа* (type constructor) (или *оператором над типами*, type operator). Он должен быть абстрактным в том же смысле, в каком абстрактными были АТД, с которыми мы работали раньше: для каждого S и T операция pair должна брать элемент S и элемент T и возвращать элемент $\text{Pair } S \ T$; fst и snd должны принимать элемент $\text{Pair } S \ T$, а возвращать, соответственно, элемент S или T ; этим должны ограничиваться сведения, известные клиенту нашей абстракции.

Исходя из этих требований, нетрудно построить сигнатуру, которую такой АТД пар должен предъявлять миру:

```
PairSig = {∃Pair\coloncolon{} *⇒*⇒*,  
          {pair: ∀X. ∀Y. X→Y→(Pair X Y),  
           fst: ∀X. ∀Y. (Pair X Y)→X,  
           snd: ∀X. ∀Y. (Pair X Y)→Y}};
```

То есть, реализация пар должна предоставлять оператор над типами Pair , а также полиморфные функции pair , fst и snd , принадлежащие требуемым типам.

Вот как можно создать пакет с таким типом:

```
pairADT =  
{*λX. λY. ∀R. (X→Y→R)→R,  
 {pair = λX. λY. λx:X. λy:Y.  
   λR. λp:X→Y→R. p x y,  
   fst = λX. λY. λp: ∀R. (X→Y→R)→R.  
             p [X] (λx:X. λy:Y. x),  
   snd = λX. λY. λp: ∀R. (X→Y→R)→R.  
             p [Y] (λx:X. λy:Y. y)}} as PairSig;
```

▷ $\text{pairADT} : \text{PairSig}$

Внутренним скрытым типом представления служит оператор $\lambda X. \lambda Y. \forall R. (X \rightarrow Y \rightarrow R) \rightarrow R$, который мы уже использовали ранее (§24.2) для представления пар. Компоненты `pair`, `fst` и `snd` в теле являются полиморфными функциями требуемых типов.

Определив АТД, мы можем его распаковать обычным способом:

```
let {Pair, pair} = pairADT
in pair.fst [Nat] [Bool] (pair.pair [Nat] [Bool] 5 true);

▷ 5 : Nat
```

30.3. Свойства исчисления

Перейдем теперь к доказательству основных свойств F_ω — в частности, как обычно, теорем о сохранении и продвижении. Идеи, лежащие в основе этих доказательств, почти такие же, как те, что мы видели раньше, но требуется соблюдать осторожность, поскольку сейчас мы работаем с довольно большой и сложной системой. В частности, потребуются заметные усилия при анализе структуры отношения эквивалентности типов. Чтобы сократить доказательства, мы будем работать только с вариантом F_ω , содержащим кванторы общности, то есть, с системой, определенной на рис. 30.1. Распространение доказательств на кванторы существования не представляет большого труда.

Базовые свойства

Для начала докажем некоторые простые свойства системы, которые понадобятся нам позднее.

Лемма 30.3.1 [УСИЛЕНИЕ]: *Если $\Gamma, x:S, \Delta \vdash T :: K$, то $\Gamma, \Delta \vdash T :: K$.*

Доказательство: В определении отношения присвоения видов нигде не упоминаются связывания термовых переменных.

Разнообразия ради, мы доказываем свойства перестановки и ослабления для F_ω совместно, а не по отдельности, как мы это делали до сих пор.

Лемма 30.3.2 [ПЕРЕСТАНОВКА И ОСЛАБЛЕНИЕ]: *Допустим, у нас есть контексты Γ и Δ , причем Δ — правильно построенная перестановка контекста Γ, Σ для некоторого Σ — то есть, Δ является перестановкой некоторого расширения Γ .*

1. *Если $\Gamma \vdash T :: K$, то $\Delta \vdash T :: K$.*
2. *Если $\Gamma \vdash t : T$, то $\Delta \vdash t : T$.*

Доказательство: Прямолинейная индукция по деревьям вывода.

Лемма 30.3.3 [Подстановка термов]: *Если $\Gamma, x:S, \Delta \vdash t : T$ и $\Gamma \vdash s : S$, то $\Gamma, \Delta \vdash [x \mapsto s]t : T$.*

Доказательство: Индукция по деревьям вывода. (УПРАЖНЕНИЕ [★]: На каком шаге используется лемма 30.3.1? А лемма 30.3.2?)

Лемма 30.3.4 [Подстановка типов]:

1. Если $\Gamma, Y :: J, \Delta \vdash T :: K$ и $\Gamma \vdash S :: J$, то $\Gamma, [Y \mapsto S]\Delta \vdash [Y \mapsto S]T :: K$.
2. Если $T \equiv U$, то $[Y \mapsto S]T \equiv [Y \mapsto S]U$.
3. Если $\Gamma, Y :: J, \Delta \vdash t : T$ и $\Gamma \vdash S :: J$, то $\Gamma, [Y \mapsto S]\Delta \vdash [Y \mapsto S]t : [Y \mapsto S]T$.

Доказательство: Прямолинейная индукция по деревьям вывода; в вариантах К-TVAR и T-VAR используется лемма об ослаблении (лемма 30.3.2). В варианте Q-APPABs требуется также заметить, что $[X \mapsto [Y \mapsto S]T_2]([Y \mapsto S]T_{12})$ совпадает с $[Y \mapsto S]([X \mapsto T_2])T_{12}$.

Эквивалентность и редукция типов

Чтобы установить свойства типизации в F_ω , удобно воспользоваться направленным вариантом отношения эквивалентности типов, которая называется *параллельной редукцией* (parallel reduction) (рис. 30.3). В отличие от отношения эквивалентности типов, здесь отсутствуют правила симметрии и транзитивности, а правило QR-APPABs разрешает редукцию в составляющих редекса. Отказ от симметрии придает отношению редукции более «вычислительный» характер, поскольку $(\lambda X :: K_{11} \cdot T_{12}) T_2$ переходит в $[X \mapsto T_2]T_{12}$, но не наоборот; при такой направленности отношение оказывается легче анализировать, например, в доказательстве леммы 30.3.12, далее в этом разделе. То, что мы отказываемся от транзитивности и позволяем компонентам проходить редукцию одновременно с редукцией лямбда-редекса — технические детали: мы вводим эти изменения ради того, чтобы выполнялось одношаговое свойство ромба, сформулированное ниже в лемме 30.3.8.

Ключевое свойство отношения параллельной редукции состоит в том, что его транзитивно-симметричное замыкание, которое обозначается символом $\Leftarrow\Rightarrow^*$, совпадает с эквивалентностью типов.

Лемма 30.3.5 $S \equiv T$ тогда и только тогда, когда $S \Leftarrow\Rightarrow^* T$.

Доказательство: В направлении (\Leftarrow) утверждение очевидно. В направлении (\Rightarrow), однако, есть единственная сложность. В то время как в выводе отношения эквивалентности типов экземпляры правил Q-SYMM и Q-TRANS могут использоваться на любом шаге, определение $\Leftarrow\Rightarrow^*$ позволяет использовать симметрию и транзитивность только на самом внешнем уровне. Справиться с этой трудностью удается, если заметить, что любой вывод $S \equiv T$ можно так преобразовать в цепочку $S = S_0 \equiv S_1 \equiv S_2 \equiv \dots \equiv S_n = T$, что каждый из этих выводов свободен от транзитивности; они «сшиваются» транзитивностью на самом верхнем уровне, причем в каждом выводе $S_i \equiv S_{i+1}$ правило Q-SYMM может использоваться (если используется вообще) только на последнем шаге.

Более того, нетрудно видеть, что отношение параллельной редукции *конфлюэнтно* (confluent, от «слияние»), как показывают следующие несколько лемм. Часто конфлюэнтность называют *свойством Чёрча-Россера* (Church-Rosser property).

Лемма 30.3.6 Если $S \Rightarrow S'$, то $[Y \mapsto S]T \Rightarrow [Y \mapsto S']T$ для любого типа T .

Доказательство: Индукция по структуре T .

Лемма 30.3.7 Если $S \Rightarrow S'$ и $T \Rightarrow T'$, то $[Y \mapsto S]T \Rightarrow [Y \mapsto S']T'$.

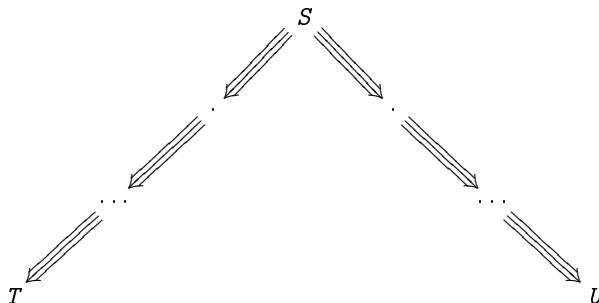
Доказательство: Индукция по второму данному выводу. В варианте QR-REFL используется лемма 30.3.6. В вариантах QR-ABS, QR-APP, QR-ARROW и QR-ALL достаточно применить предположение индукции. В варианте QR-APPABS имеем $T = (\lambda X:: K_{11} . T_{12}) T_2$ и $T' = [X \mapsto T'_2] T'_{12}$, причем $T_{12} \Rightarrow T'_{12}$ и $T_2 \Rightarrow T'_2$. Согласно предположению индукции, $[Y \mapsto S]T_{12} \Rightarrow [Y \mapsto S']T'_{12}$ и $[Y \mapsto S]T_2 \Rightarrow [Y \mapsto S']T'_2$. Применяя QR-APPABS, получаем $(\lambda X:: K_{11} . [Y \mapsto S]T_{12}) [Y \mapsto S]T_2 \Rightarrow [X \mapsto [Y \mapsto S']T'_2]([Y \mapsto S']T'_{12})$, т. е., $[Y \mapsto S](\lambda X:: K_{11} . T_{12}) T_2 \Rightarrow [Y \mapsto S']([X \mapsto T'_2] T'_{12})$.

Лемма 30.3.8 [Одношаговое свойство ромба для редукции]: Если $S \Rightarrow T$ и $S \Rightarrow U$, то существует некоторый тип V , такой, что $T \Rightarrow V$ и $U \Rightarrow V$.

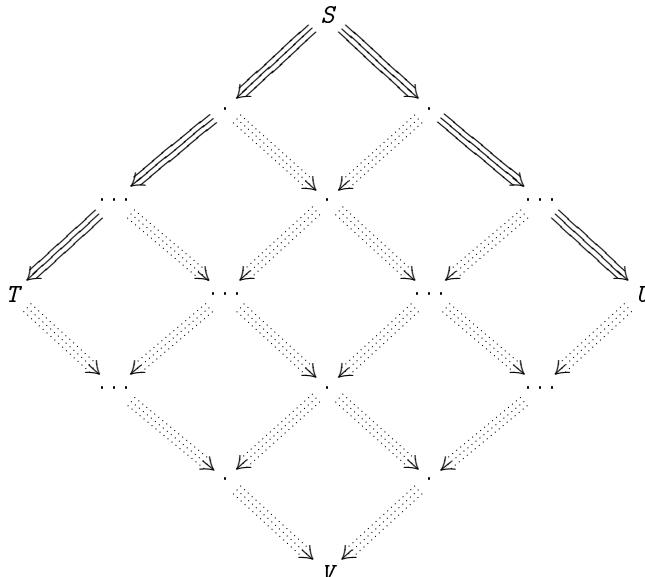
Доказательство: Упражнение [РЕКОМЕНДУЕТСЯ, ★★☆].

Лемма 30.3.9 [Конфлюэнтность]: Если $S \Rightarrow^* T$ и $S \Rightarrow^* U$, то существует некоторый тип V , такой, что $T \Rightarrow^* V$ и $U \Rightarrow^* V$.

Доказательство: Если отдельные шаги редукции от S к T и от S к U изобразить следующим образом:



то с помощью многократного применения леммы 30.3.8 можно будет заполнить внутренность диаграммы



и получить большой ромб. Нижние стороны этого ромба соответствуют обязательным редукциям.

Утверждение 30.3.10 Если $S \Leftrightarrow \Rightarrow^* T$, то существует некоторый тип U , такой, что $S \Rightarrow^* U$ и $T \Rightarrow^* U$.

Доказательство: УПРАЖНЕНИЕ [★★].

Это приводит нас к ключевому наблюдению о связи эквивалентности типов и редукции: если два типа эквивалентны, через редукцию их можно привести к общей форме. Такая структура достаточна для доказательства обсуждаемых ниже свойств инверсии.

Следствие 30.3.11 Если $S \equiv T$, то существует некоторый тип U , такой, что $S \Rightarrow^* U$ и $T \Rightarrow^* U$.

Сохранение

Мы почти готовы приступить к доказательству главного утверждения, которое гласит, что типы сохраняются при редукции. Единственное, что нам осталось перед этим доказать, — это, как обычно, лемма об инверсии (inversion lemma), которая при данном выводе типа, заключение которого имеет определенную форму, сообщает нам, как выглядят подвыходы. В свою очередь, эта лемма зависит от простого наблюдения, которое касается параллельной редукции.

Лемма 30.3.12 [СОХРАНЕНИЕ ФОРМЫ ПРИ РЕДУКЦИИ]

1. Если $S_1 \rightarrow S_2 \Rightarrow^* T$, то $T = T_1 \rightarrow T_2$, причем $S_1 \Rightarrow^* T_1$ и $S_2 \Rightarrow^* T_2$.
2. Если $\forall X:: K_1 . S_2 \Rightarrow^* T$, то $T = \forall X:: K_1 . T_2$, причем $S_2 \Rightarrow^* T_2$.

Доказательство: Прямолинейная индукция.

Лемма 30.3.13 [Инверсия]:

1. Если $\Gamma \vdash \lambda x:S_1.s_2 : T_1 \rightarrow T_2$, то $T_1 \equiv S_1$ и $\Gamma, x:S_1 \vdash s_2 : T_2$. Кроме того, $\Gamma \vdash S_1 :: *$.
2. Если $\Gamma \vdash \lambda X:: J_1.s_2 : \forall X:: K_1.T_2$, то $J_1 = K_1$ и $\Gamma, X:: J_1 \vdash s_2 : T_2$.

Доказательство: В части 1 мы доказываем по индукции следующее несколько более общее утверждение: если $\Gamma \vdash \lambda x:S_1.s_2 : S$ и $S \equiv T_1 \rightarrow T_2$, то $T_1 \equiv S_1$ и $\Gamma, x:S_1 \vdash s_2 : T_2$. Шаг индукции, правило Т-EQ, труда не представляет. Интересен базовый шаг, правило Т-ABS. Здесь S имеет вид $S_1 \rightarrow S_2$, где $\Gamma, x:S_1 \vdash s_2 : S_2$. По лемме 30.3.12(1) имеем $T_1 \equiv S_1$ и $T_2 \equiv S_2$, откуда правило Т-EQ дает нам $\Gamma, x:S_1 \vdash s_2 : T_2$. Более того, из второй предпосылки правила Т-ABS получаем $\Gamma \vdash S_1 :: *$. Часть 2 доказывается аналогично.

Теорема 30.3.14 [Сохранение]: Если $\Gamma \vdash t : T$ и $t \rightarrow t'$, то $\Gamma \vdash t' : T$.

Доказательство: Прямолинейная индукция по деревьям вывода типов. Построение доказательства аналогично доказательству теоремы о сохранении для простого типизированного лямбда-исчисления с подтипами (15.3.5).

Вариант Т-VAR: $t = x$

Не может возникнуть (правил вычисления для переменных не существует).

Вариант Т-ABS: $t = \lambda x:T_1.t_2$

Не может возникнуть (t уже является значением).

Вариант Т-APP: $t = t_1.t_2$ $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$ $\Gamma \vdash t_2 : T_{11}$ $T = T_{12}$
Из рис. 30.1 видно, что $t \rightarrow t'$ может быть выведено через три различных правила: Е-APP1, Е-APP2 и Е-APPABS. Для первых двух из них требуемый результат прямо следует из предположения индукции. Третье правило интереснее:

Подвариант Е-APPABS: $t_1 = \lambda x:S_{11}.t_{12}$ $t_2 = v_2$ $t' = [x \mapsto v_2] t_{12}$

Согласно лемме 30.3.13(1), $T_{11} \equiv S_{11}$ и $\Gamma, x:S_{11} \vdash t_{12} : T_{12}$. По правилу Т-EQ, $\Gamma \vdash t_2 : S_{11}$. Отсюда при помощи леммы о подстановке (30.3.3) получаем $\Gamma \vdash t' : T_{12}$.

Вариант Т-TABS: $t = \lambda X:: K_1.t_2$

Не может возникнуть (t уже является значением).

*Вариант Т-TAPP: $t = t_1 [T_2]$ $\Gamma \vdash t_1 : \forall X:: K_{11}.T_{12}$ $\Gamma \vdash T_2 :: K_{11}$
 $T = [X \mapsto T_2] T_{12}$*

Аналогично варианту Т-APP, но вместо леммы о подстановке термов (30.3.3) используется лемма о подстановке типов (30.3.4).

*Вариант Т-EQ: $\Gamma \vdash t : S$ $S \equiv T$ $\Gamma \vdash T :: *$*

Согласно предположению индукции, $\Gamma \vdash t' : S$. По правилу Т-EQ, $\Gamma \vdash t' : T$.

Продвижение

Следующей нашей задачей будет доказательство теоремы о продвижении. Снова большая часть необходимых для доказательства свойств у нас уже имеется — осталась лишь стандартная лемма о канонических формах, описывающая формы замкнутых значений.

Лемма 30.3.15 [КАНОНИЧЕСКИЕ ФОРМЫ]:

1. Если t — замкнутое значение и $\vdash t : T_1 \rightarrow T_2$, то t является абстракцией.
2. Если t — замкнутое значение и $\vdash t : \forall X :: K_1 . T_2$, то t является абстракцией типа.

Доказательство: Рассуждения в обеих частях леммы аналогичны; мы приводим лишь часть (1). Поскольку имеются только две формы значений — если t является значением и не является абстракцией, значит, это должна быть абстракция типа. Так что предположим (чтобы прийти к противоречию), что это абстракция типа. Тогда данное нам дерево вывода для $\vdash t : T_1 \rightarrow T_2$ должно завершаться экземпляром Т-TABS, за которым следует более одного экземпляра Т-EQ. То есть, оно должно иметь следующую форму (мы опускаем предпосылки, касающиеся видов):

$$\frac{\vdots}{\vdash t : \forall X :: K_{11} . S_{12}} \text{ (T-TABS)} \quad \frac{\forall X :: K_{11} . S_{12} \equiv U_1}{\vdash t : U_1} \text{ (T-EQ)} \\
 \vdots \\
 \frac{\vdash t : U_{n-1} \quad U_{n-1} \equiv U_n}{\vdash t : U_n} \text{ (T-EQ)} \quad U_n \equiv T_1 \rightarrow T_2 \text{ (T-TABS)}$$

Поскольку эквивалентность типов транзитивна, все эти эквивалентности мы можем слить в одну и заключить, что $\forall X :: K_{11} . S_{12} \equiv T_1 \rightarrow T_2$. Однако тогда, исходя из утверждения 30.3.10, должен существовать некоторый тип U , такой, что $\forall X :: K_{11} . S_{12} \Rightarrow^* U$ и $T_1 \rightarrow T_2 \Rightarrow^* U$. Согласно лемме 30.3.12, тип U должен в качестве внешнего конструктора одновременно иметь квантор и стрелку, что есть противоречие.

Теорема 30.3.16 [ПРОДВИЖЕНИЕ]: Допустим, имеется замкнутый правильно типизированный терм t (т. е. $\vdash t : T$ для некоторого типа T). Тогда либо t является значением, либо существует некоторый терм t' , такой, что $t \rightarrow t'$.

Доказательство: Индукция по деревьям вывода типов. Вариант Т-VAR возможен не может, поскольку t замкнут. В вариантах Т-ABS и Т-TABS утверждение теоремы следует немедленно, поскольку абстракции являются значениями. В варианте Т-EQ требуемое утверждение непосредственно следует из предположения индукции. Оставшиеся два случая — применение

и применение типа, представляют больший интерес. Мы приводим рассуждение только для случая применения типа; вариант простого применения доказывается аналогично.

Вариант Т-ТАРР: $t = t_1 [T_2] \vdash t_1 : \forall X: K_{11} . T_{12} \vdash T_2 :: K_1$

Согласно предположению индукции, либо t_1 является значением, либо он может проделать шаг вычисления. Если он может проделать шаг, то к t применимо правило Е-ТАРР. Если же терм t_1 — значение, то по лемме о канонических формах (30.3.15) нам известно, что t_1 является абстракцией типа, так что к t применимо правило Е-ТАРРТABS.

Упражнение 30.3.17 [РЕКОМЕНДУЕТСЯ, ★★]: Допустим, что мы добавили к отношению эквивалентности типов следующее странное правило:

$$T \rightarrow T \equiv \forall X: * . T$$

Какие из основных свойств системы окажутся ложными? С другой стороны, допустим, мы добавили правило:

$$S \rightarrow T \equiv T \rightarrow S$$

Какие свойства перестанут действовать в этом случае?

Присвоение видов

В определении F_ω на рис. 30.1 мы потратили некоторые усилия, чтобы убедиться в том, что все типы, которые мы можем присвоить термам согласно нашим правилам, имеют правильные виды. В частности, в правиле Т-АВС перед добавлением аннотации типа из лямбда-абстракции в контекст мы проверяем, что она правильно сформирована, а в Т-ЕQ проводится проверка, что тип T , приписываемый терму t , принадлежит виду *. Точная формулировка условия корректности, которое эти проверки обеспечивают, дана в следующем утверждении.

Определение 30.3.18 Контекст Γ называют правильно построенным (*well formed*), если (1) Γ пуст, либо (2) $\Gamma = \Gamma_1, x:T$, где Γ_1 правильно построен и $\Gamma \vdash T :: *$, либо (3) $\Gamma = \Gamma_1, X:K$ и Γ_1 правильно построен.

Утверждение 30.3.19 Если $\Gamma \vdash t : T$ и Γ правильно построен, то $\Gamma \vdash T :: *$.
Доказательство: Несложная индукция, с использованием леммы 30.3.4(1) в варианте Т-ТАРР.

Разрешимость

Из-за недостатка места мы не можем привести в этой книге полное доказательство разрешимости для F_ω — т. е. алгоритм проверки типов, а также доказательства его корректности, полноты и гарантии завершения, — но почти все требуемые для такого доказательства идеи уже известны нам из алгоритма поиска наименьших типов для Системы $F_<$, в главе 28.

Для начала заметим, что отношение присвоения видов разрешимо (поскольку его правила управляются синтаксисом). Это не удивительно, поскольку, как мы заметили, виды, по существу, являются копией простого типизированного лямбда-исчисления «на следующем уровне». Это убеждает нас в том, что проверки правильного присвоения видов в правилах типизации могут быть эффективно реализованы.

Удалим из отношения типизации единственное правило, не управляемое синтаксисом — $T\text{-EQ}$, так же, как мы удаляли $T\text{-SUB}$ из $F_{\leq\cdot}$. Рассмотрим остальные правила и выясним, какие предпосылки требуют обобщения, чтобы заменить ключевые случаи использования отсутствующего теперь правила $T\text{-EQ}$. Оказывается, что таких критических точек две.

1. В первой предпосылке правил $T\text{-APP}$ и $T\text{-TAPP}$ может потребоваться применить $T\text{-EQ}$, чтобы переписать тип левого подвыражения t_1 и вынести стрелку или квантор наружу. (Например, если в контексте переменная x связана с типом $(\lambda X. X \rightarrow X) \text{ Nat}$, то терм-применение $x 5$ имеет тип Nat только потому, что тип x может быть переписан как $\text{Nat} \rightarrow \text{Nat}$.)

Мы добиваемся этого, вводя аналог отношения выявления из §28.1. Здесь, вместо того, чтобы *расширять* наименьший тип t_1 , пока он не станет функциональным или кванторным типом, мы проводим его *редукцию* (reduction) — например, многократно применяя правила по рис. 30.3, пока дальнейшая нетривиальная редукция не станет невозможной.¹

Чтобы убедиться в том, что этот процесс придет к завершению, нужно показать, что наши правила редукции обладают свойством нормализации. Разумеется, на термах с нарушением видов редукция *не будет* нормализующей, поскольку в синтаксисе F_{ω} имеется все необходимое для того, чтобы закодировать незавершающиеся термы, такие как *омега* (с. 84). К счастью, из утверждения 30.3.19 следует, что, если мы начнем с правильно построенного контекста (и в процессе будем проводить проверки правильного видеообразования, убеждаясь в том, что каждая аннотация, вносимая нами в контекст, имеет правильный вид), то все термы, с которыми мы будем работать, будут иметь правильные виды. А для таких термов можно показать (например, модифицировав методы главы 12), что редукция всегда приводит к единственной нормальной форме.

2. Во второй предпосылке правила $T\text{-APP}$ может оказаться необходимым применить эквивалентность типов, чтобы сопоставить тип T_2 , вычисленный для терма t_2 , с типом аргумента T_{11} , стоящего перед стрелкой в функциональном типе терма t_1 . Следовательно, алгоритмический вариант этого правила должен включать *проверку* эквивалентности типов T_2 и T_{11} . Такую проверку можно реализовать, например, сведя оба этих

¹На самом деле большинство программ проверки типов для F_{ω} используют менее агрессивную разновидность редукции, известную как *слабая заголовочная редукция* (weak head reduction). При этом редуцируются только самые левые самые внешние редексы, и процесс останавливается, как только какой-то конкретный конструктор — т. е. нечто, не являющееся применением, — оказывается в начале типа.

типа к нормальным формам, а затем проверив, совпадают ли они (с точностью до имен связанных переменных).

Упражнение 30.3.20 [★★★]: Напишите программу проверки типов для F_ω на основе изложенных здесь идей. В качестве отправной точки можно использовать интерпретатор *purefsub*.

30.4. Варианты F_ω

Интуитивно понятно, что F_ω включает в себя как $\lambda\rightarrow$, так и Систему F. Это интуитивное представление можно выразить точно, определив иерархию систем F_1, F_2, F_3 и т. д. так, чтобы пределом этой иерархии была F_ω .

Определение 30.4.1 В Системе F_1 единственным видом является $*$, и не разрешается ни квантификация (\forall), ни абстракция (λ) над типами. Остальные системы определяются относительно иерархии видов уровня i (*kinds at level i*), которая выглядит следующим образом:

$$\begin{aligned}\mathcal{K}_1 &= \emptyset \\ \mathcal{K}_{i+1} &= \{*\} \cup \{J\Rightarrow K \mid J \in \mathcal{K}_i \text{ и } K \in \mathcal{K}_{i+1}\} \\ \mathcal{K}_\omega &= \bigcup_{1 \leq i} \mathcal{K}_i\end{aligned}$$

В Системе F_2 по-прежнему единственным видом является $*$, и нет лямбда-абстракции на уровне типов, но разрешена квантификация по простым типам (вида $*$). В F_3 разрешается квантификация по операторам над типами (т. е. можно выписывать выражения типа, имеющие форму $\forall X:: K. T$, где $K \in \mathcal{K}_3$), и можно вводить абстракцию над простыми типами (т. е. мы рассматриваем выражения в форме $\lambda X:: * . T$ и присваиваем им виды вроде $* \Rightarrow *$). В общем случае, F_{i+1} позволяет квантификацию над типами с видами из \mathcal{K}_{i+1} , а также абстракцию над типами с видами из \mathcal{K}_i .

F_1 представляет собой обычное простое типизированное лямбда-исчисление, $\lambda\rightarrow$. Её определение выглядит сложнее, чем рис. 9.1, поскольку включает отношения присвоения вида и эквивалентности типов. Однако и то, и другое отношение тривиально: всякий синтаксически правильный тип имеет правильный вид, а именно $*$, а единственный тип, эквивалентный T , — это сам T . Исчисление F_2 совпадает с нашей Системой F; из-за этого его часто называют *лямбда-исчислением второго порядка* (second-order lambda-calculus). F_3 — первая система, в которой отношения присвоения вида и эквивалентности типов оказываются невырожденными.

Интересно, что все программы из этой книги содержатся в F_3 . (Строго говоря, операторы над типами *Object* и *Class* из главы 32 попадают в F_4 , поскольку их аргументом служит оператор над типами вида $(* \Rightarrow *) \Rightarrow *$, но и тот, и другой можно рассматривать как сокращение, часть метаязыка, а не полноценное выражение исчисления — так мы поступали до главы 29 с *Pair*, — поскольку в примерах с *Object* и *Class* не требуется квантификация по типам этого вида.) С другой стороны, ограничение нашего языка программирования рамками F_3 вместо полного F_ω не дает никакого особенного упрощения ни

в сложности реализации, ни в метатеоретической изощренности, поскольку основные механизмы — абстракция операторов над типами и эквивалентность типов — присутствуют уже на этом уровне.

Упражнение 30.4.2 [★★★→]: *Существуют ли полезные программы, выразимые в F_4 , но невыразимые в F_3 ?*

30.5. Идем дальше: зависимые типы

В этой книге много внимания уделялось формализации разнообразных механизмов абстракции. В простом типизированном лямбда-исчислении мы формализовали операцию извлечения подтерма из терма, в результате которой получалась функция, которую затем можно конкретизировать, применяя ее к различным термам. В Системе F мы рассмотрели операцию извлечения *типа* из терма; при этом получается терм, который можно конкретизировать, применяя его к различным типам. В λ_ω мы воспроизвели механизм простого типизированного лямбда-исчисления «уровнем выше», и теперь можем извлекать подвыражение из типа и получать оператор над типами, который можно конкретизировать, применяя к различным типам.

Удобно думать об этих разновидностях абстракции в терминах *семейств выражений*, которые *индексируются* другими выражениями. Обычная лямбда-абстракция $\lambda x:T_1.t_2$ представляет собой семейство термов $[x \mapsto s]t_2$, проиндексированное термами s . Аналогично, абстракция типа $\lambda X:: K_1.t_2$ представляет собой семейство термов, проиндексированное типами, а оператор над типами есть семейство типов, индексированное типами.

$\lambda x:T_1.t_2$	семейство термов, индексированное термами
$\lambda X:: K_1.t_2$	семейство термов, индексированное типами
$\lambda X:: K_1.T_2$	семейство типов, индексированное типами.

При взгляде на этот список становится очевидно, что есть еще одна возможность, нами до сих пор не рассмотренная: семейство *типов*, индексированное *термами*. Эта форма абстракции также подробно изучена под именем *зависимых типов* (dependent types).

Зависимые типы позволяют достичь такой точности в описании поведения программ, которая идет значительно дальше всех типовых конструкций, виденных нами до сих пор. В качестве простого примера допустим, что у нас есть встроенный тип `FloatList` с обычными для такого типа операциями:

```
nil      : FloatList
cons    : Float → FloatList → FloatList
hd      : FloatList → Float
tl      : FloatList → FloatList
isnil   : FloatList → Bool
```

В языке с зависимыми типами можно уточнить простой тип `FloatList` и заменить его семейством типов `FloatList n`, каждый из которых представляет собой тип списка с n элементами.

Чтобы извлечь пользу из такого уточнения, изменим типы базовых операций над списками. Для начала припишем константе `nil` тип `FloatList 0`.

Чтобы приписать остальным операциям более точные типы, требуется усовершенствовать способ записи функциональных типов и позволить выражать *зависимость* между их аргументами и типами результатов. Например, для `cons` требуется примерно такой тип: «функция, принимающая `Float` и список длиной `n`, а возвращающая список длиной `n+1`». Если мы сделаем связывание `n` явным, введя его как дополнительный аргумент, это описание превращается в «функция, принимающая число `n`, элемент типа `Float` и список длиной `n`, и возвращающая список длиной `succ n`». Таким образом, в типе нужно отразить зависимость между *значением* первого аргумента (`n`), а также *типами* третьего аргумента (`FloatList n`) и результата (`FloatList (succ n)`). Для этого мы связываем имя с первым аргументом при помощи записи `Pn:Nat`. . . вместо `Nat → . . .`. При этом типы `cons` и других операций над списками превращаются в

```
nil      : FloatList 0
cons    : Pn:Nat. Float →FloatList n → FloatList (succ n)
hd      : Pn:Nat. FloatList (succ n) →Float
tl      : Pn:Nat. FloatList (succ n) →FloatList n
```

Типы `nil`, `cons` и `tl` точно указывают, сколько элементов содержит их результат, а `hd` и `tl` требуют в качестве аргументов непустые списки. Заметим также, что функция `isnil` больше не нужна, поскольку мы можем проверить, пуст ли элемент `FloatList n`, просто сравнив `n` с нулем.

Зависимые функциональные типы (dependent function types), имеющие форму $\Pi x:T_1.T_2$, являются более точной формой функциональных типов $T_1 \rightarrow T_2$, в которых мы связываем переменную `x`, которая представляет аргумент функции, и получаем возможность упомянуть ее в типе результата `T2`. В вырожденном случае, когда `x` не встречается в `T2`, мы записываем $\Pi x:T_1.T_2$ как $T_1 \rightarrow T_2$.

Разумеется, можно определять и новые термы с функциональными зависимыми типами. Например, функция

```
constthree = λn:Nat. λf:Float. λl:FloatList n.
              cons (succ(succ n)) f
              (cons (succ n) f
              (cons n f l));
```

которая приставляет три копии своего второго аргумента (`f`) к началу третьего аргумента (`l`), имеет тип

```
Pn:Nat. Float →FloatList n → FloatList (succ(succ(succ n))).
```

Заметим, что в каждом из трех вызовов `cons` первый аргумент имеет разное значение, что отражает различную длину аргументов-списков.

В информатике и логике имеется обширная литература по зависимым типам. Хорошими отправными точками могут служить работы Смита, Нордстрёма и Петерсона (Smith, Nordström, and Petersson, 1990), Томпсона (Thompson, 1991), Луо (Luo, 1994) и Хоффмана (Hofmann, 1997).

Упражнение 30.5.1 [★★]: *To, что тип элементов списка жестко задан как `Float`, упрощает восприятие примера. Однако нетрудно обобщить его до*

списков произвольного типа T при помощи обычновенных операторов над типами. Покажите, как это сделать.

Продолжая в том же духе, можно построить функции высшего порядка для работы со списками с аналогично уточненными типами. Например, можно написать функцию сортировки, тип которой

```
sort : Πn:Nat. FloatList n →FloatList n
```

говорит нам, что она возвращает список той же длины, что получает на входе. Более того, путем дальнейшего уточнения используемых семейств типов можно даже написать функцию `sort` так, чтобы ее тип говорил, что возвращаемый список всегда отсортирован. Проверка того, что функция `sort` принадлежит этому типу, в сущности, окажется *доказательством* (proof) того, что функция соответствует своей спецификации!

Такие примеры рисуют соблазнительную картину мира, в котором программы *верны по построению*, в котором тип программы говорит нам все, что требуется знать о ее поведении, а положительный результат работы процедуры проверки типов вселяет полную уверенность в том, что программа ведет себя как ожидается. Такое видение связано с идеей программирования посредством «извлечения вычислимого содержания» из доказательства того, что спецификация верна. Ключевое наблюдение состоит в том, что конструктивное доказательство теоремы вида «Для каждого x существует y , такое, что P » можно рассматривать как функцию, переводящую x в y , и снабженную информацией о том, что эта функция обладает свойством P . Эта информация вычислительно несущественна, т. е. представляет интерес только для процедуры проверки типов. Эти идеи исследовались в рамках проектов Nuprl (Constable et al., 1986), LEGO (Luo and Pollack, 1992; Pollack, 1994), Соq (Paulin-Mohring, 1989) и некоторых других.

К сожалению, мощность зависимых типов — это палка о двух концах. Размытие различий между проверкой типов и доказательством произвольных теорем не приводит к волшебному облегчению процесса доказательств — напротив, оно превращает проверку типов в вычислительно неразрешимую задачу! Работа математика с программами механического содействия доказательству не состоит в том, чтобы просто записать условие теоремы, нажать кнопку и ждать, пока программа ответит «да» или «нет»: требуются значительные усилия по написанию *сценариев* (proof scripts) и *тактик доказательств* (proof tactics), помогающих инструменту построить и верифицировать доказательство. Если идея правильности по построению будет доведена до логического завершения, то можно ожидать, что программисты будут тратить сопоставимые усилия на аннотацию программ подсказками и объяснениями, которые помогают программе проверки типов. Для некоторых жизненно важных программистских задач такие усилия могут быть оправданы, но для каждого дневного программирования они почти наверняка чрезмерны.

Тем не менее, предпринималось несколько попыток ввести зависимые типы в проекты практических языков программирования, в том числе Russell (Donahue and Demers, 1985; Hook, 1984), Cayenne (Augustsson, 1998), Dependent ML (Xi and Pfenning, 1998, 1999), Dependently Typed Assembly Language («Язык ассемблера с зависимыми типами», Xi and Harper, 2001),

а также *типы форм* (shape types) Джая и Секанины (Jay and Sekanina, 1997). В этих языках имеется тенденция ограничивать мощность зависимых типов различными способами. В результате получаются системы, более управляемые с вычислительной точки зрения, и для них легче автоматизировать проверку типов. Например, в языках, разработанных Xi et. al., зависимые типы используются только для статического уничтожения проверок выхода за границы массива во время выполнения; в этих языках задачи «доказательства теорем», возникающие при проверке типов, представляют собой всего лишь системы линейных ограничений, для которых существуют хорошие автоматизированные методы решения.

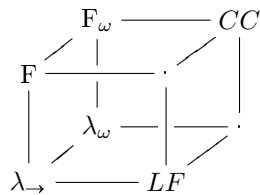
Одна из областей, в которых зависимые типы давно влияют на языки программирования — это разработка *систем модулей* (module systems), которые включают механизмы, отслеживающие *совместное использование* между модульных зависимостей. Среди достижений в этой области — язык Pebble (Burstell and Lampson, 1984), работы Маккуина (MacQueen, 1986), Митчелла и Харпера (Mitchell and Harper, 1988), Харпера и др. (Harper, Mitchell, and Moggi, 1990), а также Харпера и Стоуна (Harper and Stone, 2000). В последнее время стал использоваться технический механизм *одноэлементных видов* (singleton kinds), при котором зависимость между модулями отслеживается на уровне видов, а не типов (напр., Stone and Harper, 2000; Crary, 2000; см. также Hayashi, 1991; Aspinall, 1994).

Сочетание зависимых типов с подтипами впервые рассмотрено Карделли (Cardelli, 1988b), а затем развито и обобщено Аспиналлом (Aspinall, 1994), Пфеннингом (Pfenning, 1993b), Аспиналлом и Компаньони (Aspinall and Compagnoni, 2001), Ченом и Лонго (Chen and Longo, 1996) и Цваненбургом (Zwanenburg, 1999).

Еще одно важное приложение зависимых типов в информатике состоит в построении систем содействия доказательству и программ для автоматического доказательства теорем. В частности, простые системы типов, содержащие зависимые типы, часто называют *логическими конструкциями* (logical frameworks). Наиболее известны среди них чистое простое типизированное лямбда-исчисление с зависимыми типами, LF (Harper, Honsell, and Plotkin, 1992). LF и родственные ему системы, в частности *исчисление построений* (calculus of constructions) (Coquand and Huet, 1988; Luo, 1994), послужили основой для большого семейства сред доказательства теорем, включая AutoMath (de Bruijn, 1980), Nuprl (Constable et al., 1986), LEGO (Luo and Pollack, 1992; Pollack, 1994), Coq (Barras et al., 1997), ALF (Magnusson and Nordström, 1994) и ELF (Pfenning, 1994). Более подробный обзор можно найти у Пфеннинга (Pfenning, 1996).

Четыре формы абстракции, рассмотренные ранее в этом разделе, можно изящно изобразить на следующей диаграмме, известной как *куб Барендрегта*

(Barendregt cube):²



Все системы в этом кубе включают обыкновенную абстракцию термов. Верхняя грань представляет системы с полиморфизмом (семействами термов, проиндексированных типами), дальняя грань — системы с операторами над типами, а правая грань — системы с зависимыми типами. В правом дальнем углу находится исчисление построений (calculus of constructions, CC), которое содержит все четыре формы абстракции. Еще один упомянутый нами ранее угол — LF, простое типизированное лямбда-исчисление с зависимыми типами. Все системы куба Барендрегта, а также многие другие, могут быть представлены как частные случаи общей конструкции *систем чистых типов* (pure type systems) (Terlouw, 1989; Berardi, 1988; Barendregt, 1991, 1992; Jutting, McKinna, and Pollack, 1994; McKinna and Pollack, 1993; Pollack, 1994).

²Барендрегт (Barendregt, 1991) назвал ее *лямбда-куб* (lambda cube).

$\rightarrow \forall \Rightarrow$ Расширяет λ_ω (29.1) и Систему F (23.1)

Синтаксис	Вычисление	
$t ::=$ <i>термы:</i> x <i>переменная</i> $\lambda x:T.t$ <i>абстракция</i> $t t$ <i>применение</i> $\lambda X::K.t$ <i>абстракция типа</i> $t [T]$ <i>применение типа</i>	$t \rightarrow t'$	
$v ::=$ <i>значения:</i> $\lambda x:T.t$ <i>значение-абстракция</i> $\lambda X::K.t$ <i>значение-абстракция типа</i>	$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad (\text{E-APP1})$ $\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2} \quad (\text{E-APP2})$	
$T ::=$ <i>типы:</i> X <i>типовая переменная</i> $T \rightarrow T$ <i>тип функций</i> $\forall X::K.T$ <i>универсальный тип</i> $\lambda X::K.T$ <i>абстракция оператора</i> $T T$ <i>применение оператора</i>	$(\lambda x:T_{11}.t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APPABS})$ $\frac{t_1 \rightarrow t'_1}{t_1 [T_2] \rightarrow t'_1 [T_2]} \quad (\text{E-TAPP})$	
$\Gamma ::=$ <i>контексты:</i> \emptyset <i>пустой контекст</i> $\Gamma, x:T$ <i>связывание термовой переменной</i> $\Gamma, X::K$ <i>связывание типовой переменной</i>	$(\lambda X::K_{11}.t_{12}) [T_2] \rightarrow [X \mapsto T_2]t_{12} \quad (\text{E-TAPPTABS})$	
$K ::=$ <i>виды:</i> $*$ <i>вид простых типов</i> $K \Rightarrow K$ <i>вид операторов</i>	<i>Присвоение видов</i> $\boxed{\Gamma \vdash T :: K}$	
	$\frac{\Gamma, X::K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X::K_1.T_2 :: K_1 \Rightarrow K_2} \quad (\text{K-ABS})$	
	$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \quad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1 T_2 :: K_{12}} \quad (\text{K-APP})$	
	$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *} \quad (\text{K-ARROW})$	
	$\frac{\Gamma, X::K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X::K_1.T_2 :: *} \quad (\text{K-ALL})$	
	<i>см. на след. странице...</i>	

Рис. 30.1. Полиморфное лямбда-исчисление высших порядков (F_ω)

<i>Эквивалентность типов</i>	$S \equiv T$	<i>Типизация</i>	$\Gamma \vdash t : T$
$T \equiv T$	(Q-REFL)	$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$	(T-VAR)
$\frac{T \equiv S}{S \equiv T}$	(Q-SYMM)	$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x:T_1 \vdash t_2:T_2}{\Gamma \vdash \lambda x:T_1.t_2 : T_1 \rightarrow T_2}$	(T-ABS)
$\frac{S \equiv U \quad U \equiv T}{S \equiv T}$	(Q-TRANS)		
$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2}$	(Q-ARROW)	$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$	(T-APP)
$\frac{S_2 \equiv T_2}{\forall x :: K_1.S_2 \equiv \forall x :: K_1.T_2}$	(Q-ALL)	$\frac{\Gamma, x :: K_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x :: K_1.t_2 : \forall x :: K_1.T_2}$	(T-TABS)
$\frac{S_2 \equiv T_2}{\lambda x :: K_1.S_2 \equiv \lambda x :: K_1.T_2}$	(Q-ABS)	$\frac{\Gamma \vdash t_1 : \forall x :: K_{11}.T_{12} \quad \Gamma \vdash t_2 :: K_{11}}{\Gamma \vdash t_1 [t_2] : [x \mapsto t_2]T_{12}}$	(T-TAPP)
$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 S_2 \equiv T_1 T_2}$	(Q-APP)		
$(\lambda x :: K_{11}.T_{12}) T_2 \equiv [x \mapsto T_2]T_{12}$	(Q-APPABS)	$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T}$	(T-EQ)

Рис. 30.1. Полиморфное лямбда-исчисление высших порядков (F_ω) (*продолжение*)

$\rightarrow \forall \exists \Rightarrow$ Расширяет F_ω (30.1) и 24.1

<i>Новые синтаксические формы</i>	<i>Новые правила эквивалентности типов</i>
$T ::= \dots$	$S \equiv T$
$\{\exists X :: K, T\}$ экзистенциальный тип	
<i>Новые правила вычисления</i>	$t \rightarrow t'$
$\text{let } \{X, x\} = (\{*_T_{11}, v_{12}\} \text{ as } T_1) \text{ in } t_2$ $\rightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2$ (E-UNPACKPACK)	$\frac{S_2 \equiv T_2}{\{\exists X :: K_1, S_2\} \equiv \{\exists X :: K_1, T_2\}}$ (Q-SOME)
$\frac{t_{12} \rightarrow t'_{12}}{\{*_T_{11}, t_{12}\} \text{ as } T_1} \quad (\text{E-PACK})$ $\rightarrow \{*_T_{11}, t'_{12}\} \text{ as } T_1$	<i>Новые правила типизации</i> $\Gamma \vdash t : T$
<i>Новые правила присвоения видов</i>	$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2 \quad \Gamma \vdash \{\exists X :: K_1, T_2\} :: *}{\Gamma \vdash \{*_U, t_2\} \text{ as } \{\exists X :: K_1, T_2\} :: *}$ (T-PACK)
$\boxed{\Gamma \vdash T :: K}$	$\frac{\Gamma \vdash t_1 : \{\exists X :: K_{11}, T_{12}\} \quad \Gamma, X :: K_{11}, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2}$ (T-UNPACK)
$\frac{\Gamma, X :: K_1 \vdash T_2 :: * \quad \Gamma \vdash \{\exists X :: K_1, T_2\} :: *}{\Gamma \vdash T :: K}$ (K-SOME)	

Рис. 30.2. Экзистенциальные типы высших порядков

<i>Параллельная редукция</i>	$S \Rightarrow T$
$T \Rightarrow T$	(QR-REFL)
$\frac{S_1 \Rightarrow T_1 \quad S_2 \Rightarrow T_2}{S_1 \rightarrow S_2 \Rightarrow T_1 \rightarrow T_2}$ (QR-ARROW)	$\frac{S_2 \Rightarrow T_2}{\lambda X :: K_1 . S_2 \Rightarrow \lambda X :: K_1 . T_2}$ (QR-ABS)
$\frac{S_2 \Rightarrow T_2}{\forall X :: K_1 . S_2 \Rightarrow \forall X :: K_1 . T_2}$ (QR-ALL)	$\frac{S_1 \Rightarrow T_1 \quad S_2 \Rightarrow T_2}{S_1 \ S_2 \Rightarrow T_1 \ T_2}$ (QR-APP)
	$\frac{S_{12} \Rightarrow T_{12} \quad S_2 \Rightarrow T_2}{(\lambda X :: K_{11} . S_{12}) \ S_2 \Rightarrow [X \mapsto T_2]T_{12}}$ (QR-APPABS)

Рис. 30.3. Параллельная редукция на типах

Глава 31

Подтипы высших порядков

Последняя система, которую мы рассмотрим в этой книге, называется $F_{\zeta:}^{\omega}$ (произносится «F-sub-omega»). Она, как и $F_{\zeta:}$, представляет собой сочетание возможностей, уже исследованных нами по отдельности, — на этот раз это операторы над типами и подтипы. Можно рассматривать ее как расширение Системы $F_{\zeta:}$, лямбда-исчисления второго порядка с ограниченной квантификацией, при помощи операторов над типами. Самая интересная новая возможность $F_{\zeta:}^{\omega}$ — то, что отношение подтиповования распространяется с вида * на типы высших видов.

В литературе было предложено несколько разновидностей $F_{\zeta:}^{\omega}$, различающихся по выразительной мощности и метатеоретической сложности. Та, которую мы рассматриваем здесь, очень близка к одной из самых простых, версии Пирса и Стеффена (Pierce and Steffen, 1994). Мы не будем доказывать никаких свойств этой системы; такие доказательства можно найти у Пирса и Стеффена (Pierce and Steffen, 1994), либо у Компаньони (Compagnoni, 1994), а также у Абади и Карделли (Abadi and Cardelli, 1996), которые рассматривают схожие системы. (Чтобы представить себе длину этих доказательств, помножьте сложность главы 28 на сложность описания в §30.3.)

Основная причина, по которой мы изучаем $F_{\zeta:}^{\omega}$, — то, что эта система лежит в основе нашего последнего расширенного примера, который касается объектно-ориентированного программирования (глава 32). Этот пример не использует никаких особых хитростей определения $F_{\zeta:}^{\omega}$. Требуется только иметь возможность построения ограниченной квантификации по подтипам данного оператора над типами. Таким образом, читатель может при желании пропустить эту главу при первом чтении и вернуться к ней, если возникнут вопросы.

В этой главе изучается чистая Система $F_{\zeta:}^{\omega}$ (рис. 31.1). Соответствующая реализация называется `fomsuB` (интерпретатор `fullfomsuB` включает в себя различные расширения, например, экзистенциальные типы).

31.1. Интуитивные понятия

Взаимодействие подтипов и ограниченной квантификации с операторами над типами порождает некоторые сложности при проектировании комбинированной системы. Прежде чем приступить к формальному определению, мы кратко обсудим эти сложности.

Первый вопрос состоит в том, следует ли при наличии подтипов обобщить операторы над типами формы $\lambda X:: K_1.T_2$ до *ограниченных операторов над типами* (bounded type operators) формы $\lambda X<:T_1.T_2$. В этой главе при выборе между простотой и единобразием исчисления мы выбираем простоту, и в нашей системе будут присутствовать как ограниченная квантификация, так и *неограниченные* операторы над типами.

Второй вопрос заключается в том, как расширить отношение подтиповирования, включив в него операторы над типами. Имеется несколько вариантов. Простейший из них, который мы выбираем в этой главе — *поточечное* (pointwise) расширение отношения подтиповирования с простых типов на уровень операторов. Для абстракций мы говорим, что $\lambda X.S$ является подтиповом $\lambda X.T$ тогда, когда применение их обеих к типу U всегда дает типы, находящиеся в отношении подтиповирования. Например, $\lambda X.\text{Top} \rightarrow X$ является подтиповом $\lambda X.X \rightarrow \text{Top}$, поскольку $\text{Top} \rightarrow U$ есть подтип $U \rightarrow \text{Top}$ при любом U . Это эквивалентно тому, чтобы сказать, что $\lambda X.S$ является подтиповом $\lambda X.T$, если S будет подтиповом T в случае, если мы рассмотрим тип X как *абстрактный* (abstract), не делая никаких предположений о его подтипах и надтипах. Такая формулировка немедленно приводит к следующему правилу:

$$\frac{\Gamma, X \vdash S <: T}{\Gamma \vdash \lambda X.S <: \lambda X.T} \quad (\text{S-Abs})$$

И наоборот, если F и G являются операторами над типами и $F <: G$, то $F \cup <: G \cup$:

$$\frac{\Gamma \vdash F <: G}{\Gamma \vdash F \cup <: G \cup} \quad (\text{S-APP})$$

Заметим, что это правило срабатывает только в том случае, когда F и G применяются к *одному и тому же* аргументу U . Если известно, что F поточечно является подтиповом G , это ничего не говорит нам о том, как эти два типа поведут себя, будучи применены к разным аргументам. (В 31.4 упомянуты некоторые более сложные варианты $F<:,$ в которых этот случай принимается во внимание.)

Еще одно дополнительное правило возникает, чтобы формализовать смысл, который мы приписываем отношению эквивалентности типов. Если $S \equiv T$, то S и T содержат одни и те же элементы. Однако ясно, что типы, содержащие одни и те же элементы, являются подтипами друг друга. Отсюда возникает еще одно правило подтиповирования, в котором эквивалентность определений служит базовым случаем.

$$\frac{\Gamma \vdash S :: K \quad \Gamma \vdash T :: K \quad S \equiv T}{\Gamma \vdash S <: T} \quad (\text{S-EQ})$$

Расширив подтиповование от вида $*$ до вида $* \Rightarrow *$, можно аналогично распространить его и на более сложные виды. Например, если P и Q являются операторами над типами вида $* \Rightarrow * \Rightarrow *$, то мы говорим, что $P <: Q$ в том случае, если для любого U применение $P \ U$ является подтипом $Q \ U$ в рамках вида $* \Rightarrow *$.

Это определение имеет полезный побочный эффект: для высших видов отношения подтиповирования обладают максимальными элементами. Если мы скажем, что $\text{Top}[*] = \text{Top}$, и определим максимальные элементы высших видов:

$$\text{Top}[K_1 \Rightarrow K_2] \stackrel{\text{def}}{=} \lambda X:: K_1 . \text{Top}[K_2]$$

то с помощью несложной индукции можно показать, что $\Gamma \vdash S <: \text{Top}[K]$ (при условии, что S имеет вид K). В правилах следующего раздела мы используем этот эффект.

Переход от обычновенных ограниченных кванторов к *ограниченным кванторам высших порядков* (higher-order bounded quantifiers) не представляет труда. F_{\leq}^{ω} наследует от $F_{<}$ ограниченные кванторы вида $\forall X <: T_1 . T_2$. Обобщение этой конструкции на высшие порядки (то есть на квантификацию по операторам над типами) не требует никаких изменений в синтаксисе: мы просто говорим, что T_1 может быть любым выражением типа, включая операторы. Неограниченные кванторы высших порядков, которые мы наследуем из F_{ω} , можно рассматривать как сокращения для ограниченных кванторов с максимальным ограничением — т. е. мы считаем, что $\forall X:: K_1 . T_2$ — это сокращенная запись для $\forall X <: \text{Top}[K_1] . T_2$.

Наконец, F_{\leq}^{ω} наследует от $F_{<}$ вопрос о выборе между использованием более вычислительно доступного ядерного варианта или более выразительно мощного полного варианта для правила $S\text{-ALL}$. Мы здесь выбираем ядерный вариант; полный вариант тоже имеет смысл с семантической точки зрения, однако его метатеоретические свойства (даже те, которые, по аналогии с полной $F_{<}$, казалось бы, должны выполняться) пока не установлены.

31.2. Определения

Правила, определяющие F_{\leq}^{ω} , приведены на рис. 31.1. В определении имеется одна техническая тонкость: несмотря на то, что в системе есть две разновидности связывания для переменных типа ($X:: K$ в операторах типа и $X <: T$ в кванторах), только последняя из них может встречаться в контекстах. Когда в правилах $K\text{-Abs}$ и $S\text{-Abs}$ мы переносим связывание $X:: K$ справа от символа «штопора» налево, мы заменяем его на $X <: \text{Top}[K]$.

Еще одна тонкость состоит в том, что в F_{\leq}^{ω} исчезают правила $S\text{-REFL}$ из $F_{<}$ и $T\text{-EQ}$ из F_{ω} . Все экземпляры старого $S\text{-REFL}$ непосредственно следуют из $S\text{-EQ}$ и $Q\text{-REFL}$, а правило $T\text{-EQ}$ выводится из $T\text{-SUB}$ и $S\text{-EQ}$.

Упражнение 31.2.1 [★]: Если мы определим $Id = \lambda X. X$ и

$$\Gamma = B <: \text{Top}, A <: B, F <: Id$$

то какие из следующих утверждений о подтиповании окажутся выводимыми?

$\Gamma \vdash A$	$<: Id B$
$\Gamma \vdash Id A$	$<: B$
$\Gamma \vdash \lambda X. X$	$<: \lambda X. Top$
$\Gamma \vdash \lambda X. \forall Y <: X. Y$	$<: \lambda X. \forall Y <: Top. Y$
$\Gamma \vdash \lambda X. \forall Y <: X. Y$	$<: \lambda X. \forall Y <: X. X$
$\Gamma \vdash F B$	$<: B$
$\Gamma \vdash B$	$<: F B$
$\Gamma \vdash F B$	$<: F B$
$\Gamma \vdash \forall F <: (\lambda Y. Top \rightarrow Y). F A$	$<: \forall F <: (\lambda Y. Top \rightarrow Y). Top \rightarrow B$
$\Gamma \vdash \forall F <: (\lambda Y. Top \rightarrow Y). F A$	$<: \forall F <: (\lambda Y. Top \rightarrow Y). F B$
$\Gamma \vdash Top [* \Rightarrow *]$	$<: Top [* \Rightarrow * \Rightarrow *]$

31.3. Свойства исчисления

Доказательства основных свойств $F_{<}^\omega$, включая сохранение типов при редукции, продвижение и наличие наименьшего типа для (близкого аналога) нашей системы, можно найти в статьях, указанных в начале данной главы. Разумеется, эти доказательства должны учитывать сложности, возникающие при рассмотрении подтипов, ограниченной квантификации и операторов над типами по отдельности. Вдобавок, при попытке определения альтернативного, управляемого синтаксисом, представления правил подтиповования, возникает одна существенная новая проблема: при выводе утверждения о подтиповании может существенным образом использоваться не только правило о типовых переменных в сочетании с транзитивностью (как мы видели в §28.3), но и эквивалентность типов (правило S-EQ) вместе со свойством транзитивности.

Например, в контексте $\Gamma = X <: Top, F <: \lambda Y. Y$ утверждение $\Gamma \vdash F X <: X$ можно доказать так (игнорируя утверждения о видах):

$$\frac{\frac{\frac{\Gamma \vdash F <: \lambda Y. Y}{\Gamma \vdash F X <: (\lambda Y. Y) X} \text{ (S-APP)}}{\Gamma \vdash (\lambda Y. Y) X <: X} \text{ (S-EQ)}}{\Gamma \vdash F X <: X} \text{ (S-TRANS)}$$

Более того, заметим, что это взаимодействие правил нельзя обойти, просто приводя все выражения к нормальной форме, поскольку выражение $F A$ редуксом не является — оно становится им только тогда, когда в процессе проверки типов переменная F *расширяется* до верхней границы $\lambda Y. Y$. Решение состоит в том, чтобы нормализовывать выражения типа в начале проверки подтиповования, а затем, если потребуется, повторять нормализацию при каждой операции расширения типа.

31.4. Дополнительные замечания

Многие идеи, лежащие в основе $F_{<}^\omega$, восходят к Карделли, особенно к его статье «Структурное подтиповование и понятие степенного типа» (Cardelli,

1988a); расширение отношения подтиповирования на операторы над типами было разработано Карделли (Cardelli, 1990) и Митчеллом (Mitchell, 1990a). Ранняя семантическая модель была построена Карделли и Лонго (Cardelli and Longo, 1991), которые использовали отношение частичной эквивалентности. Компаньони и Пирс (Compagnoni and Pierce, 1996) построили модель для расширения F_{\leq}^{ω} типами-пересечениями. Более мощная модель, включающая рекурсивные типы, была предложена Брюсом и Митчеллом (Bruce and Mitchell, 1992); родственную ей модель можно найти у Абади и Карделли (Abadi and Cardelli, 1996).

Основные метатеоретические свойства приведенного здесь варианта F_{\leq}^{ω} были доказаны Пирсом и Стеффеном (Pierce and Steffen, 1994), а также независимо от них (с использованием изящного приема доказательства, который упрощает одно из основных рассуждений) в работе Компаньони (Compagnoni, 1994). Метод Компаньони также был использован Абади и Карделли (Abadi and Cardelli, 1996) для варианта F_{\leq}^{ω} , который использовал исчисление объектов, а не лямбда-исчисление в качестве ядра языка термов.

Поточечное определение подтиповирования между операторами над типами можно обобщить и разрешить подтиповирование между применением различных операторов к различным аргументам ($F S <: G T$), если мы уточним систему видов так, чтобы можно было отслеживать *полярность* (polarity) операторов над типами. Мы говорим, что оператор F *ковариантен* (covariant), если $F S <: F T$ всегда, когда $S <: T$, и *контравариантен* (contravariant), если $F T <: F S$ всегда, когда $S <: T$. Если мы введем два новых правила подтиповирования, отражающих эти свойства

$$\frac{\Gamma \vdash S <: T \quad F \text{ ковариантен}}{\Gamma \vdash F S <: F T}$$

$$\frac{\Gamma \vdash S <: T \quad F \text{ контравариантен}}{\Gamma \vdash F T <: F S}$$

то по транзитивности получим $F S <: G T$, если $F <: G$, $S <: T$ и G ковариантен. Чтобы все это работало, нам придется отмечать полярность на типовых переменных и ограничить кванторы высших порядков, чтобы они работали только с переменными определенных полярностей. Варианты F_{\leq}^{ω} с полярностью рассматривались Карделли (Cardelli, 1990), Стеффеном (Steffen, 1998), а также Дагганом и Компаньони (Duggan and Compagnoni, 1999).

Еще одно возможное расширение представленной здесь F_{\leq}^{ω} состоит в обобщении неограниченных операторов над типами $\lambda x:: K_1.T_2$ до *ограниченных операторов над типами* (bounded type operators) $\lambda x<:T_1.T_2$. Такой шаг весьма привлекателен, поскольку он повторяет то, что мы проделали, когда переходили от неограниченных к ограниченным кванторам в определении F_{\leq} : при добавлении подтипов в Систему F . С другой стороны, при этом система становится намного сложнее, поскольку требуется обобщить также систему видов и разрешить виды вроде $\forall x<:T_1.K_2$, а это, в свою очередь, создает взаимную зависимость между правилами присвоения видов и правилами подтиповирования. Распутывание этой зависимости требует немалых усилий. См. Компаньони и Гоген (Compagnoni and Goguen, 1997a,b).

Расширения F_ζ^ω , зависимыми типами исследовались Ченом и Лонго (Chen and Longo, 1996), а также Цваненбургом (Zwanenburg, 1999).

$\rightarrow \forall \Rightarrow <: \text{Top}$

Основано на F_ω (30.1) и ядерной $F_{<:}$ (26.1)

Синтаксис

$t ::=$	<i>термы:</i>
x	<i>переменная</i>
$\lambda x:T.t$	<i>абстракция</i>
$t t$	<i>применение</i>
$\lambda x <: T . t$	<i>абстракция типа</i>
$t [T]$	<i>применение типа</i>

$v ::=$	<i>значения:</i>
$\lambda x:T.t$	<i>значение-абстракция</i>
$\lambda x <: T . t$	<i>значение-абстракция типа</i>

$T ::=$	<i>типы:</i>
Top	<i>максимальный тип</i>
X	<i>типовая переменная</i>
$T \rightarrow T$	<i>тип функций</i>
$\forall x <: T . T$	<i>универсальный тип</i>
$\lambda x : K . T$	<i>абстракция оператора</i>
$T T$	<i>применение оператора</i>

$\Gamma ::=$	<i>контексты:</i>
\emptyset	<i>пустой контекст</i>
$\Gamma, x:T$	<i>связывание термовой переменной</i>
$\Gamma, x <: T$	<i>связывание типовой переменной</i>

$K ::=$	<i>виды:</i>
*	<i>вид простых типов</i>
$K \Rightarrow K$	<i>вид операторов</i>

Вычисление

$$t \rightarrow t'$$

$$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2} \quad (\text{E-APP2})$$

$$(\lambda x:T_{11}.t_{12}) \ v_2 \rightarrow [x \mapsto v_2]t_{12} \quad (\text{E-APPABS})$$

$$\frac{t_1 \rightarrow t'_1}{t_1 [T_2] \rightarrow t'_1 [T_2]} \quad (\text{E-TAPP})$$

$$(\lambda x <: T_{11} . t_{12}) [T_2] \rightarrow [x \mapsto T_2]t_{12} \quad (\text{E-TAPPTABS})$$

Присвоение видов $\boxed{\Gamma \vdash T :: K}$

$$\boxed{\Gamma \vdash \text{Top} :: *} \quad (\text{K-TOP})$$

$$\frac{X <: T \in \Gamma \quad \Gamma \vdash T :: K}{\Gamma \vdash X :: K} \quad (\text{K-TVAR})$$

$$\frac{\Gamma, X <: \text{Top}[K_1] \vdash T_2 :: K_2}{\Gamma \vdash \lambda x :: K_1 . T_2 :: K_1 \Rightarrow K_2} \quad (\text{K-ABS})$$

$$\frac{\Gamma \vdash T_1 :: K_{11} \Rightarrow K_{12} \quad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash T_1 \ T_2 :: K_{12}} \quad (\text{K-APP})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *} \quad (\text{K-ARROW})$$

$$\frac{\Gamma, X <: T_1 \vdash T_2 :: *}{\Gamma \vdash \forall x <: T_1 . T_2 :: *} \quad (\text{K-ALL})$$

см. на след. странице...

Рис. 31.1. Ограниченнная квантификация высших порядков ($F_{<:}^\omega$)

Эквивалентность типов

$$S \equiv T$$

$$T \equiv T$$

(Q-REFL)

$$T \equiv S$$

$$S \equiv T$$

(Q-SYMM)

$$\frac{S \equiv U \quad U \equiv T}{S \equiv T}$$

(Q-TRANS)

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2}$$

(Q-ARROW)

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{\forall X : S_1 . S_2 \equiv \forall X : T_1 . T_2} \quad (\text{Q-ALL})$$

$$\frac{S_2 \equiv T_2}{\lambda X : K_1 . S_2 \equiv \lambda X : K_1 . T_2} \quad (\text{Q-ABS})$$

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \ S_2 \equiv T_1 \ T_2} \quad (\text{Q-APP})$$

$$(\lambda X : K_{11} . T_{12}) . T_2 \equiv [X \mapsto T_2] T_{12} \quad (\text{Q-APPABS})$$

Подтипы

$$\Gamma \vdash S \lessdot T$$

$$\frac{\Gamma \vdash S \lessdot U \quad \Gamma \vdash U \lessdot T \quad \Gamma \vdash U :: K}{\Gamma \vdash S \lessdot T} \quad (\text{S-TRANS})$$

$$\frac{\Gamma \vdash S :: *}{\Gamma \vdash S \lessdot \text{Top}} \quad (\text{S-TOP})$$

$$\frac{\Gamma \vdash T_1 \lessdot S_1 \quad \Gamma \vdash S_2 \lessdot T_2}{\Gamma \vdash S_1 \rightarrow S_2 \lessdot T_1 \rightarrow T_2} \quad (\text{S-ARROW})$$

$$\frac{X \lessdot T \in \Gamma}{\Gamma \vdash X \lessdot T} \quad (\text{S-TVAR})$$

$$\frac{\Gamma \vdash U_1 :: K_1 \quad \Gamma, X \lessdot U_1 \vdash S_2 \lessdot T_2}{\Gamma \vdash \forall X \lessdot U_1 . S_2 \lessdot \forall X \lessdot U_1 . T_2} \quad (\text{S-ALL})$$

$$\frac{\Gamma, X \lessdot \text{Top}[K_1] \vdash S_2 \lessdot T_2}{\Gamma \vdash \lambda X :: K_1 . S_2 \lessdot \lambda X :: K_1 . T_2} \quad (\text{S-ABS})$$

$$\frac{\Gamma \vdash S_1 \lessdot T_1}{\Gamma \vdash S_1 \ U \lessdot T_1 \ U} \quad (\text{S-APP})$$

$$\frac{\Gamma \vdash S :: K \quad \Gamma \vdash T :: K \quad S \equiv T}{\Gamma \vdash S \lessdot T} \quad (\text{S-EQ})$$

Типизация

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad (\text{T-APP})$$

$$\frac{\Gamma, X \lessdot T \vdash t_2 : T_2}{\Gamma \vdash \lambda X \lessdot T . t_2 : \forall X \lessdot T . T_2} \quad (\text{T-TABS})$$

$$\frac{\Gamma \vdash t_1 : \forall X \lessdot T_{11} . T_{12}}{\Gamma \vdash t_2 \lessdot T_{11}}$$

$$\frac{\Gamma \vdash t_1 : [T_2] : [X \mapsto T_2] T_{12}}{\Gamma \vdash t_1 : T_2} \quad (\text{T-TAPP})$$

$$\frac{\Gamma \vdash t : S \quad \Gamma \vdash S \lessdot T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \quad (\text{T-SUB})$$

Глава 32

Расширенный пример: чисто функциональные объекты

В нашем последнем расширенном примере мы продолжаем исследовать *экзистенциальную модель объектов* (existential object model). Эта модель была введена в §24.2, где мы показали, как экзистенциальные пакеты можно рассматривать в качестве простых объектов. Мы также сравнили использование этого стиля абстракции и использование экзистенциальных типов для реализации более привычных абстрактных типов данных. В этой главе при помощи инструментов, описанных в предыдущих главах (операторы над типами и подтипы высших порядков), а также новой конструкции, *полиморфного обновления* (polymorphic update), описанного в §32.7, мы расширяем простые экзистенциальные объекты и получаем набор идиом, обеспечивающий гибкость полноценного объектно-ориентированного программирования, включая классы и наследование реализаций.

32.1. Простые объекты

Вспомним для начала тип чисто функциональных объектов-счетчиков, Counter:

```
Counter = { $\exists X$ , {state: $X$ , methods:{get: $X \rightarrow \text{Nat}$ , inc: $X \rightarrow X$ }}};
```

Элементами этого типа являются пакеты со скрытым *типом состояния* (state type) X , *состоянием* (state) типа X и записью, содержащей *методы*, тип которой {get: $X \rightarrow \text{Nat}$, inc: $X \rightarrow X$ }.

В нескольких ближайших разделах мы будем использовать тип {x:Nat} в качестве типа представления для всех наших объектов. (В §32.8 мы увидим, как определять объект с несколькими переменными экземпляра, а также классы, добавляющие новые переменные экземпляра.) Говоря о внутреннем типе состояния, мы будем использовать сокращение CounterR.

Примеры этой главы являются термами F_ζ^ω (рис. 31.1) с записями (11.7), числами (8.2) и полиморфными обновлениями (рис. 32.1). Соответствующая реализация на OCaml называется *fullupdate*.

```
CounterR = {x:Nat};
```

Объект-счетчик — это элемент типа `Counter`, определяемый при помощи правила введения квантора существования (T-PACK по рис. 24.1).

```
c = {*CounterR,
      {state = {x=5},
       methods = {get = λr:CounterR. r.x,
                  inc = λr:CounterR. {x=succ(r.x)}}}} as Counter;
```

```
▷ c : Counter
```

Для вызова методов объекта типа `Counter` следует этот объект распаковать, выбрать нужное поле записи методов и применить его к состоянию,

```
sendget = λc:Counter.
           let {X,body} = c in
             body.methods.get(body.state);
```

```
▷ sendget : Counter → Nat
```

И наконец, (в случае с `inc`, который должен вернуть новый объект, а не просто голое число), при необходимости результат надо перепаковать в новый объект с тем же типом представления и методами, что и в исходном объекте.

```
sendinc = λc:Counter.
           let {X,body} = c in
             {*X,
              {state = body.methods.inc(body.state),
               methods = body.methods}} as Counter;
```

```
▷ sendinc : Counter → Counter
```

При помощи этих базовых функций можно строить более сложные термы, манипулирующие объектами-счетчиками.

```
addthree = λc:Counter. sendinc (sendinc (sendinc c));
▷ addthree : Counter → Counter
```

32.2. Подтипы

Приятная возможность такого экзистенциального кодирования объектов заключается в том, что отношения подтиповирования, которые мы хотим установить между типами объектов, прямо следуют из правил подтиповирования для кванторов существования и записей. Чтобы убедиться в этом, вспомним (из рис. 26.3) правило подтиповирования для экзистенциальных типов.¹

$$\frac{\Gamma, X<:U \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X<:U, S_2\} <: \{\exists X<:U, T_2\}} \quad (\text{S-SOME})$$

¹Мы используем здесь только ядерный вариант правила; вся мощность полной версии нам не требуется. В сущности, в этой главе нам вообще не нужны *ограниченные* экзистенциальные типы — ограничением во всех кванторах существования будет Тор.

Это правило непосредственно говорит нам, что если мы определим тип объектов, у которого методов больше, чем у `Counter`, например,

```
ResetCounter =
  { $\exists X$ , {state: $X$ , methods:{get:  $X \rightarrow \text{Nat}$ , inc: $X \rightarrow X$ , reset: $X \rightarrow X$ } };
```

то он окажется подтипов `Counter`, т. е. `ResetCounter <: Counter`. Это означает, что если мы определим объект-счетчик с возможностью сброса,

```
rc = {*CounterR,
       {state = {x=0}},
       methods = {get =  $\lambda r:\text{CounterR}. r.x$ ,
                  inc =  $\lambda r:\text{CounterR}. \{x=\text{succ}(r.x)\}$ ,
                  reset =  $\lambda r:\text{CounterR}. \{x=0\}\}}}
             as ResetCounter;$ 
```

```
> rc : ResetCounter
```

то по правилу включения мы сможем передавать этот объект в функции, определенные для типа `Counter`, такие как `sendget`, `sendinc` и `addthree`:

```
rc3 = addthree rc;
```

```
> rc3 : Counter
```

Однако заметим, что при этом мы теряем информацию о типах: типом `rc3` оказывается просто `Counter`, а не `ResetCounter`.

32.3. Ограниченнaя квантификация

Разумеется, именно такого рода потеря информации из-за включения и послужила причиной введения ограниченной квантификации в главе 26. Однако ограниченная квантификация сама по себе недостаточна — для решения стоящей перед нами сейчас проблемы ее нужно обогатить некоторым дополнительным механизмом.

Чтобы понять, почему это так, рассмотрим очевидное уточнение типа `sendinc` при помощи ограниченной квантификации — $\forall C <: \text{Counter}. C \rightarrow C$. Если наш `sendinc` будет иметь такой тип, мы сможем записать `addthree` как

```
addthree =  $\lambda C <: \text{Counter}. \lambda c:C. \text{sendinc } [C] (\text{sendinc } [C] (\text{sendinc } [C] c));$ 
```

```
> addthree :  $\forall C <: \text{Counter}. C \rightarrow C$ 
```

и, применив его к `rc`, получить результат типа `ResetCounter`.

```
rc3 = addthree [ResetCounter] rc;
```

```
> rc3 : ResetCounter
```

К сожалению, у нас нет способа написать такую функцию — или, точнее, нет способа написать функцию, ведущую себя как нам надо и дать ей такой тип. Разумеется, можно написать функцию *то же самое* с таким типом,

```
wrongsendinc = λC<:Counter. λc:C. c;
▷ wrongsendinc : ∀C<:Counter. C → C
```

однако если мы попытаемся улучшить настоящую реализацию `sendinc`, добавив в начало ограниченную абстракцию типа, мы получим ошибку типизации.

```
sendinc =
  λC<:Counter. λc:C.
    let {X,body} = c in
      {*X,
       {state = body.methods.inc(body.state),
        methods = body.methods}}
      as C;
```

```
▷ Ошибка: требуется экзистенциальный тип
```

Проблема кроется в последней строке. Аннотация `as C` говорит процедуре проверки типов: «используй экзистенциальный тип `C` для создаваемого здесь пакета». Но `C` не является экзистенциальным типом — это типовая переменная. Это не просто глупое ограничение правил типизации, которые мы определили, — например, следствие того, что правила «не знают», что всякий подтип экзистенциального типа есть экзистенциальный тип. Напротив, было бы *неверно* присваивать пакету

```
{*X,
 {state = body.methods.inc(body.state),
  methods = body.methods}}
```

тип `C`. Заметим, например, что тип

```
{∃X, {state:X, methods:{get:X → Nat, inc:X → X}, junk:Bool}}
```

является подтиром `Counter`. Но приведенный нами пакет определенно не имеет такого типа: в нем нет поля `junk`. Таким образом, неверно, что для любого подтипа `C` типа `Counter` тело вышеприведенного варианта `sendinc` «на самом деле» имеет тип `C`, и только правила типизации этого не видят. Можно даже показать (например, используя денотационную модель для F_{\leq} — см. [Robinson and Tennent, 1988](#)), что в чистой F_{\leq} типы, имеющие форму $\forall C <: T. C \rightarrow C$, содержат *только* функции тождества.

Было предложено несколько способов обойти этот недостаток F_{\leq} . Один из них состоит в переходе от F_{\leq} к F_{\leq}^{ω} , используя ограниченную квантификацию высших порядков для дальнейшего уточнения типа функций вроде `sendinc`. Другая возможность — сохранить тип $\forall C <: Counter. C \rightarrow C$, но добавить в язык конструкции, позволяющие построить интересные элементы такого типа. Наконец, еще один вариант сводится к добавлению в язык ссылок. Однако этот последний путь мы уже испробовали в главе 27; в этой главе наша цель — выяснить, чего (и как) можно добиться в рамках чисто функционального стиля.

Последующее изложение сочетает два из перечисленных методов. Для разрешения проблем с квантификацией по типам объектов, замеченных нами в предыдущем разделе, используется F_{\leq}^{ω} , а похожие сложности при работе с

переменными экземпляра (§32.8) решаются через новую элементарную конструкцию для *полиморфного обновления записей* (polymorphic record update) (§32.7).

32.4. Типы интерфейсов

При помощи операторов над типами можно выразить Counter как комбинацию двух частей

```
Counter = Object CounterM;
```

где

```
CounterM = λR. {get: R → Nat, inc: R → R};
```

является оператором вида $* \Rightarrow *$, который представляет конкретный интерфейс методов для объектов-счетчиков, а

```
Object = λM::*⇒*. {∃X, {state:X, methods:M X}};
```

является оператором вида $(* \Rightarrow *) \Rightarrow *$, который отражает общую структуру всех объектных типов. При такой новой формулировке мы добиваемся отделения переменной части (интерфейса методов), в которой мы хотим разрешить подтиповование, от жесткого скелета объектов (упаковка в экзистенциальный тип и пара, состоящая из состояния и методов), в котором подтиповование не нужно, поскольку мешает переупаковке.

Чтобы достичь этого разделения, требуется ограниченная квантификация по *операторам* над типами, поскольку таким образом нам удается вытянуть интерфейс методов из типа объектов, несмотря на то, что в интерфейсе упоминается тип состояния X — переменная, связанная квантатором существования, — путем абстракции самого интерфейса методов по переменной X. Таким образом, интерфейс становится «параметрическим параметром». Такая повторная параметризация отражается как в виде Object, так и в последовательности шагов, когда упрощается применение Object CounterM: сначала CounterM подставляется в тело Object, и получается

```
{∃X, {state:X, methods:(λR. {get: R → Nat, inc: R → R}) X)}
```

а затем проводится подстановка X в тело CounterM, и получается

```
{∃X, {state:X, methods:{get:X → Nat, inc:X → X}}}.
```

Если мы аналогично разделим ResetCounter,

```
ResetCounterM = λR. {get: R → Nat, inc: R → R, reset: R → R};  
ResetCounter = Object ResetCounterM;
```

то получим не только, как выше, что

```
ResetCounter <: Counter
```

но и что

```
ResetCounterM <: CounterM
```

согласно правилам о подтиповании для операторов над типами. Таким образом, разделение объектных типов на обобщенный каркас и специализированный интерфейс дает нам осмысленное понятие *подтипование интерфейсов* (interface subtyping), отдельное от отношений подтипования между конкретными типами объектов.

Подтипование интерфейсов тесно связано, как концептуально, так и технически, с идеей *соответствия* (matching), введенной Брюсом и др. (Bruce, Petersen, and Fiech, 1997) и затем исследованной Абади и Карделли (Abadi and Cardelli, 1995, 1996).

32.5. Отправка сообщений объектам

Теперь можно исправить сломанную версию `sendinc` из §32.3, проведя абстракцию по *подинтерфейсам* (subinterfaces) `CounterM` вместо *подтипов* `Counter`.

```
sendinc =
  λM<:CounterM. λc:Object M.
    let {X, b} = c in
      {*X,
       {state = b.methods.inc(b.state),
        methods = b.methods}}
    as Object M;

▷ sendinc : ∀M<:CounterM. Object M → Object M
```

С интуитивной точки зрения, тип `sendinc` можно прочитать так: «дайте мне интерфейс объекта, расширяющий интерфейс счетчика, затем дайте объект с таким интерфейсом, и я верну вам другой объект с таким же интерфейсом».

Упражнение 32.5.1 [★]: Почему эта версия `sendinc`, в отличие от предыдущей, правильно типизирована?

Чтобы вызывать методы объектов-счетчиков и счетчиков собросом, мы конкретизируем полиморфные функции вызова методов соответствующей сигнатурой интерфейса, `CounterM` либо `ResetCounterM` (мы предполагаем, что `sendget` и `sendreset` определены аналогично).

```
sendget [CounterM] (sendinc [CounterM] c);

▷ 6 : Nat

sendget [ResetCounterM]
  (sendreset [ResetCounterM]
    (sendinc [ResetCounterM] rc));

▷ 0 : Nat
```

Упражнение 32.5.2 [РЕКОМЕНДУЕТСЯ, ★★]: Определите `sendget` и `sendreset`.

32.6. Простые классы

Начнем рассмотрение классов, как мы делали в гл. 18, с *простых классов* без `self`.

В §18.6 мы определили простой класс (для императивного кодирования объектов, которые представляли собой записи из методов) как функцию, которая переводит состояния в объекты. Этот способ позволял порождать множество объектов, каждый из которых имел одинаковый набор методов, но свой набор свежесозданных переменных экземпляра. В этой главе «объект» есть нечто большее, чем простой набор методов: он также имеет тип представления и состояние. С другой стороны, поскольку мы работаем в рамках чисто функциональной модели, то каждый из этих методов принимает состояние в качестве параметра (и, при необходимости, возвращает объект с обновленным состоянием), так что не нужно передавать состояние классу в момент создания объекта. В сущности, здесь класс можно рассматривать как простую запись из методов (по-прежнему предполагая, что все объекты используют один и тот же тип представления):

```
counterClass =
  {get = λr:CounterR. r.x,
   inc = λr:CounterR. {x=succ(r.x)}}
as {get: CounterR → Nat, inc:CounterR → CounterR};

▷ counterClass : {get: CounterR → Nat, inc:CounterR → CounterR}
```

или, используя оператор `CounterM` для сокращения аннотации,

```
counterClass =
  {get = λr:CounterR. r.x,
   inc = λr:CounterR. {x=succ(r.x)}}
as CounterM CounterR;

▷ counterClass : CounterM CounterR
```

Мы создаем экземпляры таких классов, указывая начальное состояние и упаковывая его вместе с методами (т. е., с классом) в объект.

```
c = {*CounterR,
      state = {x=0},
      methods = counterClass}
as Counter;

▷ c : Counter
```

Определение подкласса сводится к построению новой записи с методами; при этом часть полей копируется из уже существующей записи.

```
resetCounterClass =
  let super = counterClass in
  {get = super.get,
   inc = super.inc,
   reset = λr:CounterR. {x=0}}
```

```

as ResetCounterM CounterR;

▷ resetCounterClass : ResetCounterM CounterR

```

Чтобы обобщить эти простые классы и добиться правильной работы в примерах, приведенных в конце главы 18, нужны еще две вещи: возможность добавлять в подклассах новые переменные экземпляра и поддержка `self`. В следующих двух разделах мы справимся с первой из этих задач; в финальном §32.9 решается вопрос работы с `self`.

32.7. Полиморфные обновления

Чтобы получить возможность добавлять к классам переменные экземпляра, нам нужно ввести еще один новый механизм — элементарную конструкцию, которая проводит *полиморфное обновление* (polymorphic update) полей записи, а также соответственно уточнить типы записей. Нужда в этих новшествах возникает оттого, что, разрешая изменение числа переменных экземпляра в разных классах, мы делаем надклассы полиморфными по отношению к переменным экземпляра их подклассов. Посмотрим, как это происходит.

Предположим, мы хотим определить подкласс `resetCounterClass`, добавив к нему метод `backup`, сохраняющий текущее значение счетчика, и заставив метод `reset` возвращаться к этому сохраненному значению, а не к жестко заданной константе. Чтобы сохранить это значение, нам нужно расширить тип представления с `{x:Nat}` до `{x:Nat, old:Nat}`. Такая разница представлений немедленно создает техническую сложность. Возможность использовать метод `inc` из класса `resetCounterClass` при определении `BackupCounterClass` зависит от того, ведет ли код себя одинаково в обоих классах. Однако если наборы переменных различны, то и поведение получается различным: `inc` в типе `ResetCounter` ожидает состояние типа `{x:Nat}` и возвращает новое состояние того же типа, тогда как `inc` в типе `BackupCounter` ожидает и порождает состояния типа `{x:Nat, old:Nat}`.

Чтобы справиться с этой сложностью, достаточно заметить, что методу `inc` на самом деле не нужно знать, что тип состояния *равен* `{x:Nat}` либо `{x:Nat, old:Nat}`; ему нужно только знать, что состояние *содержит* переменную `x`. Другими словами, можно унифицировать оба этих метода, если дать им тип `$\forall S <: \{x:\text{Nat}\}. S \rightarrow S$` .

Теперь с полями возникает та же самая проблема, что и с целыми объектами в §32.3: в нашем теперешнем языке тип `$\forall S <: \{x:\text{Nat}\}. S \rightarrow S$` содержит только функцию тождества. Чтобы справиться с этим, нам снова нужен некий механизм, допускающий более точную форму ограниченной квантификации; в данном случае, проще всего добавить механизм для *полиморфного обновления* (polymorphic update) полей в записях.² Если `r` является записью с полем

²Как и раньше, существует несколько способов добиться похожего эффекта — либо вводя различные примитивы (некоторые из которых перечислены в разделе §32.10), либо используя полиморфизм, как в Pierce and Turner, 1994 — вариант более тяжеловесный с точки зрения нотации, но более элементарный теоретически. Тот вариант, который используется здесь, выбран из-за своей простоты, а также потому, что он естественным образом подходит к нижеприведенным примерам.

х типа T, а t — терм типа T, то выражение $r \leftarrow x = t$ будет означать «запись, совпадающая с r, за исключением поля x, значение которого равно t». Заметим, что такая операция обновления вполне функциональна — она не изменяет r, а создает копию записи с другим значением поля x.

С помощью такого примитива обновления записей можно примерно так написать функцию, отражающую ожидаемое поведение тела метода inc:

```
f = λX<:{a:Nat}. λr:X. r ← a = succ(r.a);
```

Нужно, однако, соблюдать осторожность. Наивное правило типизации для операции обновления выглядит так:

$$\frac{\Gamma \vdash r : R \quad \Gamma \vdash R <: \{l_j:T_j\} \quad \Gamma \vdash t : T_j}{\Gamma \vdash r \leftarrow l_j = t : R}$$

Однако, оно некорректно. Например, допустим, что у нас есть

```
s = {x={a=5,b=6},y=true};
```

Поскольку $s : \{x:\{a:Nat,b:Nat\},y:Bool\}$, и при этом $\{x:\{a:Nat,b:Nat\},y:Bool\} <: \{\{a:Nat\}\}$, вышеприведенное правило позволило бы нам написать

```
s ← x={a=8} : {x:{a:Nat,b:Nat},y:Bool}
```

что неверно, поскольку $s \leftarrow x = \{a=8\}$ редуцируется до $\{x=\{a=8\},y=true\}$.

Проблема возникла из-за подтиповирования в глубину для поля x, что позволило вывести $\{x:\{a:Nat,b:Nat\},y:Bool\} <: \{x:\{a:Nat\}\}$. Подтиповирование в глубину требуется запретить для полей, которые могут быть обновлены. Этого можно добиться ценой простого расширения, позволяющего метить поля особым знаком # («решетка»).

Правила для таких «обновляемых записей» и для самой операции обновления приведены на рис. 32.1. Мы усложняем синтаксис типов записей так, что каждое поле оказывается снабжено меткой *изменчивости* (variance), указывающей, разрешено ли для этого поля подтиповирование в глубину — метка # его запрещает, а пустая строка разрешает (такой выбор означает, что непомеченные записи будут вести себя как раньше). Правило подтиповирования в глубину, S-RCDDEPTH, уточняется и позволяет теперь подтиповирование только для непомеченных полей. Наконец, мы добавляем правило подтиповирования S-RCDVARIANCE, позволяющее изменять метки полей с # на пустую строку — иными словами, «забывать», что какое-то поле может обновляться. Правило типизации для операции обновления требует, чтобы заменяемое поле было помечено знаком #. Правило E-UPDATE реализует операцию обновления.

Теперь функция f, обсуждавшаяся выше, пишется так:

```
f = λX<:{#a:Nat}. λr:X. r ← a = succ(r.a);
```

```
> f : ∀X<:{#a:Nat}. X → X
```

и используется так:

```
r = {#a=0, b=true};
f [{#a:Nat,b:Bool}] r;
```

$\rightarrow \forall <: \text{Top} \{ \} \leftarrow$

Основано на $F_<$: (26.1) с записями (11.7)

Новые синтаксические формы

$t ::= \dots$ *термы:*
 $\{ \iota_i l_i = t_i^{i \in 1..n} \}$ *запись*
 $t \leftarrow l = t$ *обновление поля*

$T ::= \dots$ *типы:*
 $\{ \iota_i l_i : T_i^{i \in 1..n} \}$ *тип записей*

$\iota ::= \#$ *инвариантное (обновляемое) поле*
 пусто *ковариантное (фиксированное) поле*

Новые правила вычисления

$t \rightarrow t'$

$$\begin{aligned} &\{ \iota_j l_j = v_j^{j \in 1..n} \} \leftarrow l_i = v \\ \rightarrow &\{ \iota_j l_j = v_j^{j \in 1..i-1}, \iota_i l_i = v, \iota_k l_k = v_k^{k \in i+1..n} \} \\ &\quad (\text{E-UPDATEV}) \\ &\{ \iota_i l_i = v_i^{i \in 1..n} \} . l_j \rightarrow v_j \\ &\quad (\text{E-PROJRCDF}) \end{aligned}$$

$$\begin{aligned} &t_j \rightarrow t'_j \\ &\{ \iota_i l_i = v_i^{i \in 1..j-1}, \iota_j l_j = t_j, \iota_k l_k = t_k^{k \in j+1..n} \} \\ \rightarrow &\{ \iota_i l_i = v_i^{i \in 1..j-1}, \iota_j l_j = t'_j, \iota_k l_k = \dots \} \\ &\quad (\text{E-RCDF}) \end{aligned}$$

Новые правила подтипования

$\Gamma \vdash S <: T$

$$\Gamma \vdash \{ \iota_i l_i : T_i^{i \in 1..n+k} \} <: \{ \iota_i l_i : T_i^{i \in 1..n} \} \\ (\text{S-RCDDWIDTH})$$

для каждого i , $\Gamma \vdash S_i <: T_i$

если $\iota_i = \#$, то $\Gamma \vdash T_i <: S_i$

$$\Gamma \vdash \{ \iota_i l_i : S_i^{i \in 1..n} \} <: \{ \iota_i l_i : T_i^{i \in 1..n} \} \\ (\text{S-RCDDDEPTH})$$

$$\Gamma \vdash \{ \dots \# l_i : S_i \dots \} <: \{ \dots l_i : S_i \dots \} \\ (\text{S-RCDDVARIANCE})$$

Новые правила типизации

$\Gamma \vdash t : T$

$$\frac{\text{для каждого } i, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{ \iota_i l_i = t_i^{i \in 1..n} \} : \{ \iota_i l_i : T_i^{i \in 1..n} \}} \\ (\text{T-RCDF})$$

$$\frac{\Gamma \vdash t_1 : \{ \iota_i l_i : T_i^{i \in 1..n} \}}{\Gamma \vdash t_1 . l_j : T_j} \\ (\text{T-PROJ})$$

$$\frac{\Gamma \vdash r : R \quad \Gamma \vdash R <: \{ \# l_j : T_j \} \quad \Gamma \vdash t : T_j}{\Gamma \vdash r \leftarrow l_j = t : R} \\ (\text{T-UPDATE})$$

Рис. 32.1. Полиморфное обновление

$\triangleright \{ \# a = 1, b = \text{true} \} : \{ \# a : \text{Nat}, b : \text{Bool} \}$

Корректность операции обновления основывается на следующем наблюдении об уточненном отношении подтипования:

Утверждение 32.7.1 Если $\vdash R <: \{ \# l : T_1 \}$, то $R = \{ \dots \# l : R_1 \dots \}$, причем $\vdash R_1 <: T_1$ и $\vdash T_1 <: R_1$.

Упражнение 32.7.2 [РЕКОМЕНДУЕТСЯ, ★ ★ ★]: Соблюдается ли в этом исчислении свойство минимальной типизации? Если да, докажите это. Если нет, покажите, как можно исправить ситуацию.

32.8. Добавление переменных экземпляра

При помощи конструкций из предыдущего раздела мы можем написать `counterClass`, полиморфный по отношению к типу своего внутреннего состояния.

```
CounterR = {#x:Nat};

counterClass =
  λR<:CounterR.
    {get = λs:R. s.x,
     inc = λs:R. s←x=succ(s.x)}
    as CounterM R;

▷ counterClass : ∀R<:CounterR. CounterM R
```

Чтобы построить объект нашего нового класса `counterClass`, мы просто задаем `CounterR` в качестве типа представления:

```
c = {*CounterR,
      {state = {#x=0},
       methods = counterClass [CounterR]}}
    as Object CounterM;

▷ c : Counter
```

Заметим, что объекты, построенные с помощью нового класса, имеют тот же самый тип `Counter = Object CounterM`, что и построенные нами ранее: изменения в обработке переменных состояния целиком скрыты внутри классов. Старые функции вызова методов также можно использовать с объектами, созданными при помощи новых классов.

В том же стиле мы можем написать и `resetCounterClass`.

```
resetCounterClass =
  λR<:CounterR.
    let super = counterClass [R] in
      {get = super.get,
       inc = super.inc,
       reset = λs:R. s←x=0}
      as ResetCounterM R;
▷ resetCounterClass : ∀R<:CounterR. ResetCounterM R
```

Наконец, теперь мы можем написать `backupCounterClass`, на этот раз абстрагируясь по подтипаам `BackupCounterR` (что и было целью всего нашего упражнения).

```
backupCounterM = λR. {get:R→Nat, inc:R→R, reset:R→R, backup:R→R};
```

```

backupCounterR = {#x:Nat,#old:Nat};
backupCounterClass =
  λR<:BackupCounterR.
    let super = resetCounterClass [R] in
      {get = super.get,
       inc = super.inc,
       reset = λs:R. s←x=s.old,
       backup = λs:R. s←old=s.x}
    as BackupCounterM R;
▷ backupCounterClass : ∀R<:BackupCounterR. BackupCounterM R;

```

32.9. Классы с переменной self

В §18.9 мы рассмотрели способ расширения императивных классов механизмом, который позволяет методам класса рекурсивно вызывать друг друга. Это расширение имеет смысл и в чисто функциональном окружении.

Для начала абстрагируем `counterClass` по набору методов `self`, подходящих для того же типа представления `R`.

```

counterClass =
  λR<:CounterR.
  λself: Unit →CounterM R.
  λ_:Unit.
    {get = λs:R. s.x,
     inc = λs:R. s←x=succ(s.x)}
   as CounterM R;

```

Как и в §18.9, с помощью аргумента `Unit` мы откладываем вычисление во время операции `fix`, создающей методы объекта. Тип `self` включает соответствующую абстракцию по `Unit`.

Чтобы построить объект на основе этого класса, возьмем неподвижную точку функции `counterClass` и применим ее к `unit`.

```

c = {*CounterR,
      {state = {#x=0},
       methods = fix (counterClass [CounterR]) unit}}
   as Object CounterM;

▷ c : Counter

```

Теперь определим подкласс с операцией `set`, обладающий следующим интерфейсом:

```
SetCounterM = λR. {get: R →Nat, set:R → Nat → R inc:R → R};
```

Реализация `setCounterClass` определяет метод `set` и использует методы `set` и `get` из `self` при реализации своего метода `inc`:

```

setCounterClass =
  λR<:CounterR.
  λself:Unit →SetCounterM R.
  λ_:Unit.

```

```

let super = counterClass [R] self unit in
{get = super.get,
 set = λs:R. λn:Nat. s←x=n,
 inc = λs:R. (self unit).set s (succ((self unit).get s))}
as SetCounterM R;

```

Наконец, объединяя все механизмы этой главы, мы можем построить подкласс счетчиков с отслеживанием обращений, в которых операция `set` считает, сколько раз она была вызвана.

```

InstrCounterM =
  λR. {get: R → Nat, set:R → Nat → R, inc:R →
R, accesses:R → Nat};

InstrCounterR = {#x:Nat, #count:Nat};

instrCounterClass =
  λR<:InstrCounterR.
  λself: Unit → InstrCounterM R.
  λ_:Unit.
  let super = setCounterClass [R] self unit in
  {get = super.get,
   set = λs:R. λn:Nat.
     let r = super.set s n in
     r←count=succ(r.count),
   inc = super.inc,
   accesses = λs:R. s.count}
  as InstrCounterM R;

```

Обратите внимание, что вызовы `inc` включаются в счетчик обращений, поскольку `inc` реализован в терминах метода `set` переменной `self`.

Для проверки создадим объект-счетчик с отслеживанием обращений и пошлем ему несколько сообщений.

```

ic = {*InstrCounterR,
       {state = {#x=0, #count=0},
        methods = fix (instrCounterClass [InstrCounterR]) unit}}
as Object InstrCounterM;

> ic : Object InstrCounterM

sendaccesses [InstrCounterM] (sendinc [InstrCounterM] ic);

> 1 : Nat

```

Упражнение 32.9.1 [РЕКОМЕНДУЕТСЯ ★ ★★]: *Определите подкласс `instrCounterClass`, добавляющий методы `backup` и `reset`.*

32.10. Дополнительные замечания

Первая «чисто функциональная» интерпретация объектов в рамках типизированного лямбда-исчисления была основана на рекурсивно определяемых

записях; ее впервые предложил Карделли (Cardelli, 1984), а затем множество ее вариантов изучали Кэймин и Редди (Reddy, 1988; Kamin and Reddy, 1994), Кук и Палсберг (Cook and Palsberg, 1989), а также Митчелл (Mitchell, 1990а). Бестиповая форма этой модели вполне эффективно использовалась в денотационной семантике бестиповых объектно-ориентированных языков. Ее типизированная форма пригодна для кодирования отдельных объектно-ориентированных примеров, но при систематической интерпретации типизированных объектно-ориентированных языков она вызывает трудности. Наиболее успешным проектом в этом отношении явились работы Кука и его соавторов (Cook, Hill, and Canning, 1990; Canning, Cook, Hill, and Olthoff, 1989а; Canning, Cook, Hill, Olthoff, and Mitchell, 1989б).

Пирс и Тёрнер (Pierce and Turner, 1994) ввели кодирование, опирающееся только на систему типов с экзистенциальными типами, без рекурсивных типов. Это привело Хоффмана и Пирса (Hofmann and Pierce, 1995б) к первой единообразной управляемой типами интерпретации объектов в рамках функционального исчисления. На той же конференции Брюс представил работу (Bruce, 1994) по семантике функционального объектно-ориентированного языка. Эта семантика сначала была построена как прямое отображение в денотационную модель F_ζ^ω , но недавно ее переформулировали как способ кодирования объектов, зависящий одновременно от экзистенциальных и рекурсивных типов. Тем временем, будучи разочарованы трудностями кодирования объектов в рамках лямбда-исчисления, Абади и Карделли представили исчисление, в котором объекты служат примитивным элементом (Abadi and Cardelli, 1996). Однако позднее Абади, Карделли и Вишванатан (Abadi, Cardelli, and Viswanathan, 1996) открыли адекватный способ кодирования этого исчисления в терминах ограниченных экзистенциальных и рекурсивных типов. Обзор этих достижений можно найти у Брюса и др. (Bruce, Cardelli, and Pierce, 1999), а также у Абади и Карделли (Abadi and Cardelli, 1996).

Кодирование объектов, применяемое в этой главе, было расширено на случай *множественного наследования* (multiple inheritance) (классов, имеющих более одного надкласса) Компаньони и Пирсом (Compagnoni and Pierce, 1996). Основная техническая идея состоит в добавлении в F_ζ^ω *типов-пересечений* (intersection types) (§15.7).

Имеется множество предложений по устранению недостатков чистой ограниченной квантификации второго порядка, которые мы рассмотрели в §32.3. Помимо двух решений, использованных нами в этой главе, — ограниченной квантификации высших порядков и элементарной конструкции для полиморфного обновления записей — есть еще несколько других стилей полиморфного обновления (Cardelli and Mitchell, 1991; Cardelli, 1992; Fisher and Mitchell, 1996; Poll, 1996), а также *структурное развертывание* (structural unfolding) рекурсивных типов (Abadi and Cardelli, 1996), *позитивное подтиповование* (positive subtyping) (Hofmann and Pierce, 1995а), *полиморфная перепаковка* (polymorphic repacking) экзистенциальных типов (Pierce, 1996) и *деструкторы типов* (type destructors) (Hofmann and Pierce, 1998).

Другой подход, основанный на *полиморфизме строчных переменных* (row variable polymorphism), изучается в работах Ванда (Wand, 1987, 1988, 1989б), Реми (Rémy, 1990, 1989; Rémy, 1992), Вуйона (Vouillon, 2000, 2001) и дру-

гих. Он также лежит в основе объектно-ориентированных конструкций OCaml (Rémy and Vouillon, 1998).

Начни сначала, — важно
ответил Король, — и
продолжай, пока не дойдешь до
конца. Как дойдешь — кончай!

Льюис Кэрролл,
пер. Н. Демуровой

Приложения

Приложение А

Решения избранных упражнений

3.2.4. РЕШЕНИЕ: $|S_{i+1}| = |S_i|^3 + |S_i| \times 3 + 3$, а $|S_0| = 0$. Таким образом, $|S_3| = 59439$.

3.2.5. РЕШЕНИЕ: Простого доказательства по индукции достаточно. При $i = 0$ утверждение очевидно. При $i = j + 1$, для некоторого $j \geq 0$, мы предполагаем, что $S_j \subseteq S_i$, и должны доказать, что $S_i \subseteq S_{i+1}$ — а именно, для каждого терма $t \in S_i$ требуется показать, что $t \in S_{i+1}$. Итак, допустим, что $t \in S_i$. Поскольку S_i определяется как объединение трех множеств, нам известно, что t имеет одну из трех возможных форм:

1. $t \in \{\text{true}, \text{false}, 0\}$. В этом случае очевидно, что $t \in S_{i+1}$, по определению S_{i+1} .
2. $t = \text{succ } t_1, \text{pred } t_1$ или $\text{iszero } t_1$ для некоторого $t_1 \in S_j$. Поскольку по предположению индукции $S_j \subseteq S_i$, мы имеем $t_1 \in S_i$, и, таким образом, по определению S_{i+1} , $t \in S_{i+1}$.
3. $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$, где $t_1, t_2, t_3 \in S_j$. Опять же, поскольку, по предположению индукции, $S_j \subseteq S_i$, имеем $t \in S_{i+1}$, по определению S_{i+1} .

3.3.4. РЕШЕНИЕ: (Мы приводим доказательство только для принципа индукции по глубине; остальные доказательства строятся аналогично.) Нам известно, что для любого терма s , если $P(r)$ выполняется для всех r , чья глубина меньше, чем глубина s , то выполняется и $P(s)$; требуется показать, что $P(s)$ выполняется для всех s . Определим новый предикат Q на натуральных числах:

$$Q(n) = \forall s, \text{ где } \text{depth}(s) = n. P(s)$$

Теперь при помощи индукции на натуральных числах (2.4.2) мы доказываем, что $Q(n)$ выполняется для всех n .

3.5.5. РЕШЕНИЕ: Пусть P — предикат на деревьях вывода утверждений о вычислениях.

Если для каждого дерева вывода \mathcal{D} ,

предполагая $P(\mathcal{C})$ для всех непосредственных поддеревьев \mathcal{C} ,
мы можем доказать $P(\mathcal{D})$,

то $P(\mathcal{D})$ выполняется для всех \mathcal{D} .

3.5.10. РЕШЕНИЕ:

$$\frac{\frac{\frac{t \rightarrow t'}{t \rightarrow^* t'}}{t \rightarrow^* t''}}{t \rightarrow^* t''}$$

3.5.13. РЕШЕНИЕ: (1) 3.5.4 и 3.5.11 перестают работать. 3.5.7, 3.5.8 и 3.5.12 сохраняются. (2) В этом случае не выполняется только 3.5.4; остальные теоремы остаются истинными. Во втором варианте интересно то, что, несмотря на потерю детерминированности одношагового вычисления в результате добавления нового правила, окончательный результат многошагового вычисления по-прежнему определяется однозначно: все дороги ведут в Рим. Доказательство этого утверждения не так уж сложно, хоть и не столь тривиально, как раньше. Главное наблюдение состоит в том, что одношаговое отношение вычисления обладает так называемым *свойством ромба* (diamond property):

Лемма A.1 [Свойство ромба]

Если $r \rightarrow s$ и $r \rightarrow t$, где $s \neq t$, то существует терм u , такой, что $s \rightarrow u$ и $t \rightarrow u$.

Доказательство: Из правил вычисления ясно, что описываемая ситуация может возникнуть, только если r имеет вид *if* r_1 *then* r_2 *else* r_3 . Доказательство ведется по индукции на паре деревьев вывода, которые используются для вывода утверждения $r \rightarrow s$ и $r \rightarrow t$, с анализом вариантов для последнего шага в каждом выводе.

Вариант i:

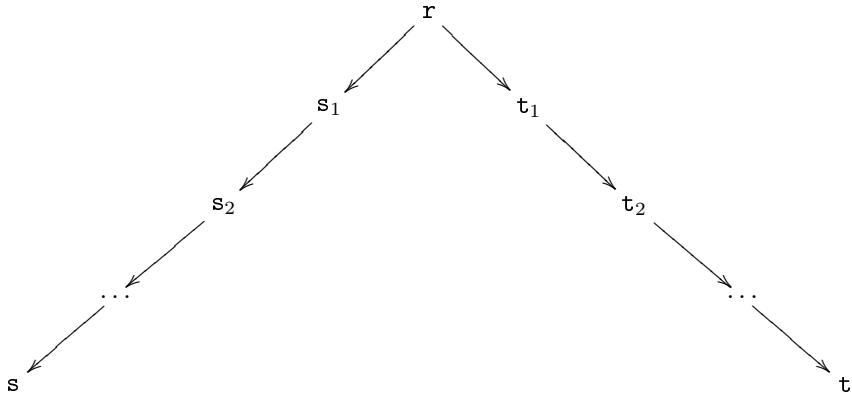
Допустим, $r \rightarrow s$ по E-IFTRUE, а $r \rightarrow t$ по E-FUNNY2. Тогда, исходя из формы этих правил, мы знаем, что $s = r_2$, и что $t = \text{if true then } r'_2 \text{ else } r_3$, где $r_2 \rightarrow r'_2$. Но тогда выбор $u = r'_2$ дает требуемое, поскольку мы знаем, что $s \rightarrow r'_2$, и видим, что $t \rightarrow r'_2$ по правилу E-IFTRUE.

Вариант ii:

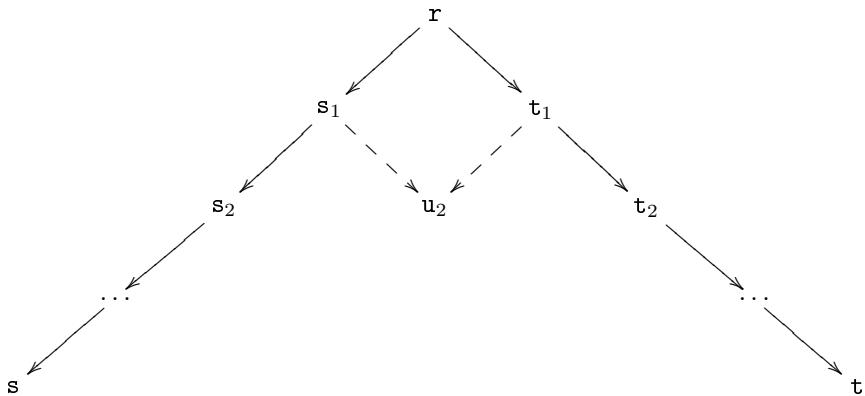
Допустим, что E-IF является последним правилом в выводе как $r \rightarrow s$, так и $r \rightarrow t$. Из формы E-IF мы знаем, что s имеет вид *if* r'_1 *then* r_2 *else* r_3 , а t имеет вид *if* r''_1 *then* r_2 *else* r_3 , где $r_1 \rightarrow r'_1$ и $r_1 \rightarrow r''_1$. Но тогда, по предположению индукции, имеется терм r'''_1 такой, что $r'_1 \rightarrow r'''_1$ и $r''_1 \rightarrow r'''_1$. В таком случае мы можем завершить рассуждение, взяв $u = \text{if } r'''_1 \text{ then } r_2 \text{ else } r_3$ и заметив, что $s \rightarrow u$ и $t \rightarrow u$ по правилу E-IF.

Для остальных вариантов доказательство строится похожим образом.

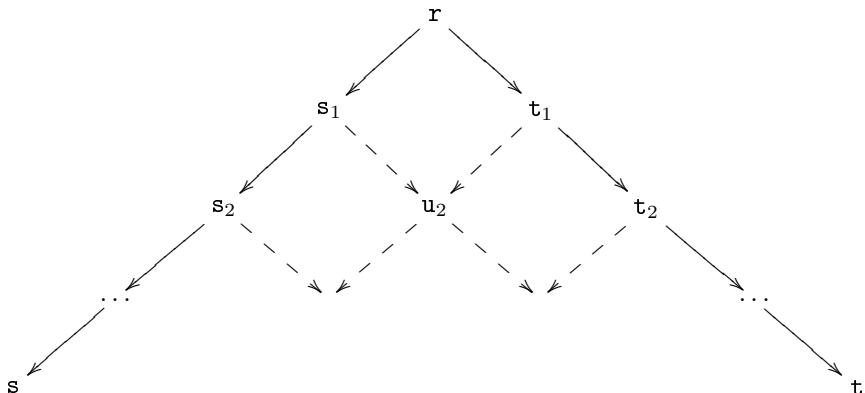
Доказательство единственности результатов теперь можно провести стандартным методом «стягивания диаграммы». Допустим, $r \rightarrow^* s$ и $r \rightarrow^* t$.



Тогда, используя лемму A.1, мы можем «стянуть вместе» s_1 и t_1



затем стянуть попарно s_2 с u_2 и u_2 с t_2



и действовать так до тех пор, пока s и t не будут стянуты вместе, и из многих одношаговых ромбов не получится один большой многошаговый.

Отсюда следует, что, если при вычислении r могут получиться значения v и w , то они должны совпадать (как v , так и w являются нормальными формами, так что при вычислении они могут давать один и тот же терм, только если они одинаковы с самого начала).

3.5.14. РЕШЕНИЕ: Индукция по структуре t .

Вариант: t — значение.

Поскольку все значения являются нормальными формами, такая ситуация возникнуть не может.

Вариант: $t = \text{succ } t_1$.

Рассматривая правила вычисления, замечаем, что единственное из них, при помощи которого можно вывести $t \rightarrow t'$ и $t \rightarrow t''$ — это правило E-SUCC (во всех остальных правилах внешний конструктор левой части отличается от **succ**). Таким образом, должны существовать два поддерева вывода с заключениями вида $t_1 \rightarrow t'_1$ и $t_1 \rightarrow t''_1$. По предположению индукции (которое применимо здесь, поскольку t_1 — подтерм t), имеем $t'_1 = t''_1$. Но тогда $\text{succ } t'_1 = \text{succ } t''_1$, что нам и требуется.

Вариант: $t = \text{pred } t_1$.

В этом случае есть три различных правила вычисления (E-PRED, E-PREDZERO и E-PREDSUCC), которые могли бы быть использованы при редукции t к t' и t'' . Заметим, однако, что правила эти не перекрываются: если t соответствует левой части одного из них, то оно определенно *не* соответствует левым частям остальных. (Например, если t соответствует E-PRED, значит, t_1 точно не является значением, в частности, оно не равно 0 или **succ v**.) Это дает нам основание считать, что для вывода $t \rightarrow t'$ и $t \rightarrow t''$ использовалось *одно и то же* правило. Если правило было E-PRED, мы, как и в предыдущем варианте, используем предположение индукции. Если это было E-PREDZERO или E-PREDSUCC, то результат следует непосредственно.

Остальные варианты:

Аналогично.

3.5.16. РЕШЕНИЕ: Пусть метапеременная t обозначает терм нового вида, включающего в себя **wrong** (а также термы, где **wrong** встречается как подвыражение), а g обозначает терм исходного вида, в котором **wrong** встретиться не может. Мы будем использовать запись $t \xrightarrow{w} t'$, которая обозначает новое отношение вычисления, к которому добавлены правила для **wrong**, а $g \xrightarrow{o} g'$ — для старой формы вычисления. Теперь мы можем точно сформулировать утверждение о том, что два варианта семантики соответствуют друг другу: всякий терм (исходного вида), вычисление которого по старой семантике заходит в тупик, в новой семантике сведется к **wrong**, и наоборот. Формально:

Утверждение A.2 Для всех термов исходного вида, g , (1) $g \xrightarrow{o}^* v$ тогда и только тогда, когда $g \xrightarrow{w}^* v$, и (2) ($g \xrightarrow{o}^* g'$, где g' находится в тупике) тогда и только тогда, когда $(g \xrightarrow{w}^* \text{wrong})$.

Доказательство состоит из нескольких шагов. Прежде всего, заметим, что новые добавленные правила не отменяют теорему 3.5.14:

Лемма А.3 Расширенное отношение вычисления детерминированно.

Это означает, что во всех случаях, когда g за один шаг переводится в g' в исходной семантике, оно может перевестись в g' и в расширенной семантике; более того, g' — единственный терм, в который g может перейти в новой семантике.

На следующем этапе покажем, что если терм (уже) оказался в тупике в исходной семантике, то в расширенной семантике его вычисление приведет к *wrong*.

Лемма А.4 Если g — тупиковый терм, то $g \xrightarrow{w^*} \text{wrong}$.

Доказательство: индукция по структуре g .

Вариант: $g = \text{true}, \text{false}$ или 0

Такого быть не может (мы предположили, что g тупиковый).

Вариант: $g = \text{if } g_1 \text{ then } g_2 \text{ else } g_3$

Поскольку g тупиковый, g_1 должен находиться в нормальной форме (иначе можно было бы применить E-IF). Понятно, что g_1 не может быть *true* или *false* (иначе были бы применимы E-IF-TRUE либо E-IFFALSE, и g не был бы тупиковым). Рассмотрим оставшиеся варианты:

Подвариант: $g_1 = \text{if } g_{11} \text{ then } g_{12} \text{ else } g_{13}$

Поскольку g_1 находится в нормальной форме и при этом, очевидно, не является значением, он в тупике. Предположение индукции говорит нам, что $g_1 \xrightarrow{w^*} \text{wrong}$. Отсюда мы можем построить вывод $g \xrightarrow{w^*} \text{if wrong then } g_2 \text{ else } g_3$. Добавляя к этому выводу экземпляр правила E-IF-WRONG, получаем $g \xrightarrow{w^*} \text{wrong}$, что и требуется.

Подвариант: $g_1 = \text{succ } g_{11}$

Если g_{11} является числовым значением, то g_1 представляет собой *badbool*, и правило E-IF-WRONG немедленно приводит к $g \xrightarrow{w} \text{wrong}$. В противном случае, по определению, g_1 находится в тупике, и предположение индукции говорит нам, что $g_1 \xrightarrow{w^*} \text{wrong}$. На основании этого вывода можно построить вывод $g \xrightarrow{w^*} \text{if wrong then } g_2 \text{ else } g_3$. Добавляя к нему экземпляр E-IF-WRONG, получаем $g \xrightarrow{w^*} \text{wrong}$.

Другие подварианты:

Аналогично.

Вариант: $g = \text{succ } g_1$

Поскольку g в тупике, мы знаем (из определения значений), что g_1 находится в нормальной форме, но не является числовым значением. Есть две возможности: либо g_1 равен *true* или *false*, либо g_1 тоже не является значением, то есть находится в тупике. В первом случае, E-SUCC-WRONG немедленно приводит к $g \xrightarrow{w} \text{wrong}$; во втором предположение индукции дает нам $g_1 \xrightarrow{w^*} \text{wrong}$, и мы рассуждаем, как раньше.

Остальные варианты:

Аналогично.

Леммы A.3 и A.4 вместе дают нам направление «и только тогда» (\Rightarrow) части (2) утверждения A.2. Чтобы доказать обратное направление, требуется продемонстрировать, что терм, который в расширенной семантике «вот-вот даст неправильное значение», в обычной семантике является тупиковым.

Лемма A.5 *Если $g \xrightarrow{w} t$ в расширенной семантике, и t содержит в качестве подтерма $wrong$, то в исходной семантике g находится в тупике.*

Доказательство: несложная индукция на (расширенных) деревьях вывода вычислений.

Сочетая это утверждение с леммой A.3, мы получаем направление «тогда» (\Leftarrow) утверждения A.2, часть (2), и доказательство закончено.

3.5.17. Решение: Мы доказываем два направления «тогда и только тогда» по отдельности, в утверждениях A.7 и A.9. В каждом случае, мы начинаем с доказательства технической леммы, которая устанавливает некоторое полезное свойство вычисления с большим/малым шагом.

Лемма A.6 *Если $t_1 \rightarrow^* t'_1$, то $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow^* \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$ (и аналогично для других конструкторов).*

Доказательство: Простая индукция.

Утверждение A.7 *Если $t \Downarrow v$, то $t \rightarrow^* v$.*

Доказательство: Индукция по выводу $t \Downarrow v$, с разбором вариантов по последнему использованному правилу.

Вариант B-VALUE: $t = v$

Утверждение следует непосредственно.

Вариант B-IFTTRUE: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
 $t_1 \Downarrow \text{true}$
 $t_2 \Downarrow v$

По предположению индукции, $t_1 \rightarrow^ \text{true}$. По лемме A.6*

$\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow^* \text{if true then } t_2 \text{ else } t_3$

По правилу E-IFTTRUE $\text{if true then } t_2 \text{ else } t_3 \rightarrow^ t_2$. По предположению индукции $t_2 \rightarrow^* v$. Нужный результат следует из транзитивности отношения \rightarrow^* .*

Остальные случаи аналогичны.

Лемма A.8 *Если*

$\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow^* v$

то либо $t_1 \rightarrow^ \text{true}$ и $t_2 \rightarrow^* v$, либо $t_1 \rightarrow^* \text{false}$ и $t_3 \rightarrow^* v$. Более того, последовательности вычисления для t_1 и t_2 или t_3 строго короче, чем исходная последовательность вычисления. (То же верно и для других конструкторов.)*

Доказательство: Индукция по длине исходной последовательности вычисления. Поскольку условное выражение не является значением, в последовательности должен быть по крайней мере один шаг. Проведем разбор вариантов последнего правила, использованного на этом шаге (это должно быть одно из IF-правил).

Вариант E-IF: $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow^* \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \rightarrow^* v$
 $t \rightarrow t'_1$

По предположению индукции либо $t'_1 \rightarrow^* \text{true}$ и $t_2 \rightarrow^* v$, либо $t'_1 \rightarrow^* \text{false}$ и $t_3 \rightarrow^* v$. Добавление начального шага $t_1 \rightarrow t'_1$ к выводу $t'_1 \rightarrow^* \text{true}$ или $t'_1 \rightarrow^* \text{false}$ дает необходимый результат. Легко проверить, что получающиеся последовательности вычисления короче исходных.

Вариант E-IFTTRUE: $\text{if true then } t_2 \text{ else } t_3 \rightarrow^* v$
Утверждение следует немедленно.

Вариант E-IFFFALSE: $\text{if false then } t_2 \text{ else } t_3 \rightarrow^* v$
Утверждение следует немедленно.

Утверждение A.9 Если $t \rightarrow^* v$, то $t \Downarrow v$.

Доказательство: Индукция по числу шагов вычисления (с малым шагом) при выводе $t \rightarrow^* v$.

Если $t \rightarrow^* v$ за 0 шагов, то $t = v$, и результат следует из B-VALUE. В противном случае требуется анализ вариантов формы t .

Вариант: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

По лемме A.8 либо (1) $t_1 \rightarrow^* \text{true}$ и $t_2 \rightarrow^* v$, либо (2) $t_1 \rightarrow^* \text{false}$ и $t_3 \rightarrow^* v$. Доказательство в обоих случаях аналогично, так что предположим, для определенности, вариант (1). Лемма A.8 утверждает также, что последовательности вычисления для $t_1 \rightarrow^* \text{true}$ и $t_2 \rightarrow^* v$ короче, чем для t , так что применимо предположение индукции; это дает нам $t_1 \Downarrow \text{true}$ и $t_2 \Downarrow v$. Отсюда мы можем вывести $t \Downarrow v$ через правило B-IFTTRUE.

Другие конструкторы термов разбираются аналогично.

4.2.1. РЕШЕНИЕ: Каждый раз, когда eval вызывает себя рекурсивно, на стек вызовов помещается обработчик try. Поскольку на каждом шаге вычисления происходит один рекурсивный вызов, в конце концов стек переполнится. В сущности, из-за того, что рекурсивный вызов eval обернут в try, он перестает быть хвостовым, хотя и продолжает так выглядеть. Более правильная (хотя и более трудная для чтения) версия eval такова:

```
let rec eval t =
  let t'opt = try Some(eval1 t) with NoRuleApplies →
    None in
  match t'opt with
    Some(t') → eval t'
  | None → t
```

5.2.1. РЕШЕНИЕ:

```
or = λb. λc. b tru c;  
not = λb. b fls tru;
```

5.2.2. РЕШЕНИЕ:

```
scc2 = λn. λs. λz. n s (s z);
```

5.2.3. РЕШЕНИЕ:

```
times2 = λm. λn. λs. λz. m (n s) z;
```

Или же, более компактно:

```
times3 = λm. λn. λs. m (n s);
```

5.2.4. РЕШЕНИЕ: Опять же, существует несколько вариантов решения:

```
power1 = λm. λn. m (times n) c1;  
power2 = λm. λn. m n;
```

5.2.5. РЕШЕНИЕ:

```
subtract1 = λm. λn. n prd n;
```

5.2.6. РЕШЕНИЕ: Вычисление $\text{prd } c_n$ занимает $O(n)$ шагов, поскольку prd строит последовательность из n пар чисел, а затем выбирает первый элемент последней пары.**5.2.7.** РЕШЕНИЕ: Вот простое решение:

```
equal = λm. λn.  
    and (iszro (m prd n))  
        (iszro (n prd m));
```

5.2.8. РЕШЕНИЕ: Вот решение, которое имел в виду я:

```
nil = λc. λn. n;  
cons = λh. λt. λc. λn. c h (t c n);  
head = λl. l (λh. λt. h) fls;  
tail = λl.  
    fst (l (λx. λp. pair (snd p) (cons x (snd p)))  
        (pair nil nil));  
isnil = λl. l (λh. λt. fls) tru;
```

А вот несколько другой подход:

```
nil = pair tru tru;  
cons = λh. λt. pair fls (pair h t);  
head = λz. fst (snd z);  
tail = λz. snd (snd z);  
isnil = fst;
```

5.2.9. РЕШЕНИЕ: Мы используем `if`, а не `test`, чтобы избежать ситуации, в которой всегда вычисляются обе ветви условного выражения, что привело

бы к расхождению `factorial`. Чтобы избежать такого расхождения при использовании `test`, нужно защитить обе ветви, завернув их в пустые лямбда-абстракции. Поскольку абстракции являются значениями, наша стратегия с вызовом по значению внутрь них не заглядывает, а передает их как есть в `test`, а уже он выбирает одну из них и отдает обратно. Тогда мы применяем все выражение `test` к аргументу-заглушке, скажем, `c0`, чтобы заставить выбранную ветвь вычисляться.

```
ff = λf. λn.
      test
        (iszro n) (λx. c1) (λx. (times n (f (prd n))) c0);
factorial = fix ff;
equal c6 (factorial c3);

▷ (λx. λy. x)
```

5.2.10. РЕШЕНИЕ: Вот рекурсивное решение:

```
cn = λf. λm. if iszero m then c0 else scc (f (pred m));
churchnat = fix cn;
```

Простая проверка его работоспособности:

```
equal (churchnat 4) c4;

▷ (λx. λy. x)
```

5.2.11. РЕШЕНИЕ:

```
ff = λf. λl.
      test (isnil l)
        (λx. c0) (λx. (plus (head l) (f (tail l))) c0;
sumlist = fix ff;

l = cons c2 (cons c3 (cons c4 nil));
equal (sumlist l) c9;

▷ (λx. λy. x)
```

Разумеется, функцию суммирования списка можно написать и без использования `fix`:

```
sumlist' = λl. l plus c0;
equal (sumlist' l) c9;

▷ (λx. λy. x)
```

5.3.3. РЕШЕНИЕ: Индукция по размеру `t`. Предположим, что требуемое свойство выполняется для всех термов, меньших `t`; нужно доказать его для самого `t`. Нужно рассмотреть три варианта:

Вариант: t = x

Следует непосредственно: $|FV(t)| = |\{x\}| = 1 = \text{size}(t)$.

Вариант: $t = \lambda x.t_1$

Согласно предположению индукции, $|FV(t_1)| \leq size(t_1)$. Считаем так:

$$|FV(t)| = |FV(t_1) \setminus \{x\}| \leq |FV(t_1)| \leq size(t_1) < size(t).$$

Вариант: $t = t_1 \ t_2$

Согласно предположению индукции, $|FV(t_1)| \leq size(t_1)$ и $|FV(t_2)| \leq size(t_2)$.

Считаем так:

$$|FV(t)| = |FV(t_1) \cup FV(t_2)| \leq |FV(t_1)| + |FV(t_2)| \leq size(t_1) + size(t_2) < size(t).$$

5.3.6. РЕШЕНИЕ: Для полной (недетерминистской) бета-редукции правила та-ковы:

$$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{t_1 \ t_2 \rightarrow t_1 \ t'_2} \quad (\text{E-APP2})$$

$$\frac{t_1 \rightarrow t'_1}{\lambda x.t_1 \rightarrow \lambda x.t'_1} \quad (\text{E-ABS})$$

$$(\lambda x.t_{12}) \ t_2 \rightarrow [x \mapsto t_2]t_{12} \quad (\text{E-APPABS})$$

(Заметим, что синтаксическая категория значений не используется.) Для нормального порядка вычислений один из вариантов записи правил таков:

$$\frac{na_1 \rightarrow na'_1}{na_1 \ t_2 \rightarrow na'_1 \ t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \rightarrow t'_2}{nanf_1 \ t_2 \rightarrow nanf_1 \ t'_2} \quad (\text{E-APP2})$$

$$\frac{t_1 \rightarrow t'_1}{\lambda x.t_1 \rightarrow \lambda x.t'_1} \quad (\text{E-ABS})$$

$$(\lambda x.t_{12}) \ t_2 \rightarrow [x \mapsto t_2]t_{12} \quad (\text{E-APPABS})$$

где синтаксические категории нормальной формы, нормальной формы-не абстракции и не-абстракции определяются так:

nf ::= *нормальные формы:*

$\lambda x.nf$
 $nanf$

nanf ::= *нормальные формы-не абстракции:*

x
 $nanf \ nf$

na ::= *не-абстракции:*

x
 $t_1 \ t_2$

(Это определение по сравнению с остальными выглядит немножко неестественно. Обычно нормальный порядок редукции определяется так: «как при полной бета-редукции, но при условии, что всегда выбирается только самый левый, самый внешний редекс».)

Ленивая стратегия определяет значения как произвольные абстракции, аналогично стратегии вызова по значению. Правила вычисления таковы:

$$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \quad (\text{E-APP1})$$

$$(\lambda x. t_{12}) \ t_2 \rightarrow [x \mapsto t_2] t_{12} \quad (\text{E-APPABS})$$

5.3.8. РЕШЕНИЕ:

$$\lambda x. t \Downarrow \lambda x. t$$

$$\frac{t_1 \Downarrow \lambda x. t_{12} \quad t_2 \Downarrow v_2 \quad [x \mapsto v_2] t_{12} \Downarrow v}{t_1 \ t_2 \Downarrow v}$$

6.1.1. РЕШЕНИЕ:

```
c0 = λ. λ. 0;
c2 = λ. λ. 1 (1 0);
plus = λ. λ. λ. λ. 3 1 (2 0 1);
fix = λ. (λ. 1 (λ. (1 1) 0)) (λ. 1 (λ. (1 1) 0));
foo = (λ. (λ. 0)) (λ. 0);
```

6.1.5. РЕШЕНИЕ: Эти две функции можно определить так:

$\text{removenames}_\Gamma(x)$	=	индекс самого правого x в последовательности Γ
$\text{removenames}_\Gamma(\lambda x. t_1)$	=	$\lambda. \text{removenames}_{\Gamma,x}(t_1)$
$\text{removenames}_\Gamma(t_1 \ t_2)$	=	$\text{removenames}_{\Gamma,x}(t_1) \text{removenames}_{\Gamma,x}(t_2)$
$\text{restorenames}_\Gamma(k)$	=	k -е имя в Γ
$\text{restorenames}_\Gamma(\lambda. t_1)$	=	$\lambda x. \text{restorenames}_{\Gamma,x}(t_1)$ где x — первое имя переменной, не входящее в $\text{dom}(\Gamma)$
$\text{restorenames}_\Gamma(t_1 \ t_2)$	=	$\text{restorenames}_{\Gamma,x}(t_1) \text{restorenames}_{\Gamma,x}(t_2)$

Требуемые свойства функций removenames и restorenames доказываются простой структурной индукцией на термах.

6.2.2. РЕШЕНИЕ:

$$1. \lambda. \lambda. 1 (0 4)$$

$$2. \lambda. 0 3 (\lambda. 0 1 4)$$

6.2.5. РЕШЕНИЕ:

$[0 \mapsto 1] (0 (\lambda.x.\lambda.y.a))$	$=$	$1 (\lambda.x.\lambda.y.a)$
	т. е.	$a (\lambda.x.\lambda.y.a)$
$[0 \mapsto 1 (\lambda.x.\lambda.y.a)] (0 (\lambda.z.(a z)))$	$=$	$(1 (\lambda.x.\lambda.y.a)) (\lambda.z.(a z))$
	т. е.	$(a (\lambda.z.a)) (\lambda.z.(a z))$
$[0 \mapsto 1] (\lambda.x.0 z)$	$=$	$\lambda.x.0 z$
	т. е.	$\lambda.b.b a$
$[0 \mapsto 1] (\lambda.x.0 z)$	$=$	$\lambda.x.0 z$
	т. е.	$\lambda.a'.a a'$

6.2.8. РЕШЕНИЕ: Пусть имеется контекст именования Γ . Обозначим через $\Gamma(x)$ индекс переменной x в Γ , считая справа. Требуемое свойство формулируется так:

$$\text{removenames}_\Gamma([x \mapsto s]t) = [\Gamma(x) \mapsto \text{removenames}_\Gamma(s)](\text{removenames}_\Gamma(t))$$

Доказательство строится индукцией по t , при помощи определений 5.3.5 и 6.2.4, некоторых несложных вычислений, а также пары простых лемм, доказывающих свойства *removenames* и других простых операций с термами. При рассмотрении абстракции ключевую роль играет соглашение 5.3.4.

6.3.1. РЕШЕНИЕ: Единственный случай, в котором мог бы возникнуть отрицательный индекс, — это когда в сдвигаемом терме встретилась переменная 0. Однако этого случиться не может, поскольку мы только что провели подстановку вместо переменной 0 (при этом терм, подставляемый вместо 0, только что был сдвинут вверх, и в нем тоже не может быть вхождений переменной 0).

6.3.2. РЕШЕНИЕ: Доказательство эквивалентности представлений, использующих индексы и уровни, можно найти в Lescanne and Rouyer-Degli, 1995. Уровни де Брауна обсуждаются также в de Bruijn, 1972 и Filinski, 1999, раздел 5.2.

8.3.5. РЕШЕНИЕ: Нет: отказ от этого правила приводит к потере свойства продвижения. Если мы действительно против того, чтобы определять предшественника 0, то с ним надо бороться как-то иначе — например, вызывая исключение (глава 14), если программа пытается его вычислить, или переопределив тип *pred* так, чтобы его можно было применять только к строго положительным числам — с помощью типов-пересечений (§15.7) или зависимых типов (§30.5).

8.3.6. РЕШЕНИЕ: Вот контрпример: терм (*if false then true else 0*) не имеет типа, но при вычислении дает правильно типизированный терм 0.

8.3.7. РЕШЕНИЕ: Свойство сохранения типов для семантики с большим шагом похоже на то, которое мы привели для семантики с малым шагом: если правильно типизированный тип при вычислении дает некоторое окончательное значение, то тип этого значения будет таким же, как у исходного терма. Доказательство аналогично нашему. С другой стороны, свойство продвижения делает намного более сильное утверждение: всякий правильно типизированный терм дает при вычислении некоторое окончательное значение — то есть, вычисление на правильно типизированных термах всегда завершается. Для арифметических выражений это утверждение оказывается верным; однако, для многих более интересных языков (языков с рекурсией общего вида, см. §11.11) оно выполняться не будет. В таких языках свойство продвижения

просто *отсутствует*: в сущности, нет никакого способа отличить ошибочное состояние от зацикливания. Это одна из причин, почему теоретики языков программирования обычно предпочитают стиль с малым шагом.

Другая альтернатива состоит в том, чтобы явно указывать в семантике с большим шагом переходы в состояние `wrong`, как в упражнении 8.3.8. Этот стиль используют, например, Абади и Карделли в операционной семантике для своего исчисления объектов (Abadi and Cardelli, 1996, с. 87).

8.3.8. РЕШЕНИЕ: В расширенной семантике тупиковые состояния вообще отсутствуют — всякий терм, не являющийся значением, либо переходит в другой терм обычным образом, либо явно превращается в `wrong` (это, разумеется, тоже надо *доказать*), — так что свойство продвижения тривиально. С другой стороны, теорема о редукции субъекта дает нам больше информации, чем раньше. Поскольку у `wrong` никакого типа нет, утверждение о том, что правильно типизированный терм может перейти только в другой правильно типизированный терм, означает, в частности, что он не может перейти во `wrong`. В сущности, доказательство старой теоремы о продвижении становится частью нового доказательства сохранения.

9.2.1. РЕШЕНИЕ: Потому что множество выражений типа пустое (в синтаксисе типов нет базового варианта).

9.2.3. РЕШЕНИЕ: Пример такого контекста:

$$\Gamma = f : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}, x:\text{Bool}, y:\text{Bool}$$

В общем случае, годится любой контекст вида

$$\Gamma = f:S \rightarrow T \rightarrow \text{Bool}, x:S, y:T$$

где S и T — произвольные типы. Рассуждения такого рода играют ключевую роль в алгоритме *реконструкции типов* (type reconstruction) из главы 22.

9.3.2. РЕШЕНИЕ: Предположим, что терм x имеет тип T , и сведем это предположение к противоречию. По лемме об инверсии, левый подтерм (x) должен иметь тип $T_1 \rightarrow T_2$, а правый подтерм (тоже x) должен иметь тип T_1 . Согласно варианту леммы об инверсии, относящемуся к переменным, как $x : T_1 \rightarrow T_2$, так и $x : T_1$, должны следовать из предположений в Γ . Поскольку у x в Γ может быть только *одно* связывание, имеем $T_1 \rightarrow T_2 = T_1$. Но это невозможно — все типы имеют конечный размер, так что тип не может служить под выражением самого себя. Мы получили требуемое противоречие.

Заметим, что если бесконечные выражения для типов были бы разрешены, мы *могли бы* найти решение для уравнения $T_1 \rightarrow T_2 = T_1$. К этому вопросу мы вернемся в главе 20.

9.3.3. РЕШЕНИЕ: Предположим, что $\Gamma \vdash t : S$ и $\Gamma \vdash t : T$. С помощью индукции по выводу $\Gamma \vdash t : T$ покажем, что $S = T$.

Вариант T-VAR: $t = x$

где $x:T \in \Gamma$

Согласно пункту 1 леммы об инверсии (9.3.1), последнее правило во всяком дереве вывода типа $\Gamma \vdash t : S$ также должно быть T-VAR, и $S = T$.

Вариант T-Abs: $t = \lambda y:T_2. t_1$
 $T = T_2 \rightarrow T_1$
 $\Gamma, y:T_2 \vdash t_1 : T_1$

Согласно пункту 2 леммы об инверсии, последнее правило во всяком дереве вывода $\Gamma \vdash t : S$ также должно быть T-Abs, и в этом дереве вывода должно иметься поддерево с выводом $\Gamma, y:T_2 \vdash t_1 : S_1$, причем $S = T_2 \rightarrow S_1$. По предположению индукции (относительно поддерева с заключением $(\Gamma, y:T_2 \vdash t_1 : T_1)$) получаем $S_1 = T_1$, откуда немедленно следует $S = T$.

Варианты T-APP, T-TRUE, T-FALSE, T-IF: Аналогично.

9.3.9. РЕШЕНИЕ: Индукция по дереву вывода $\Gamma \vdash t : T$. На каждом шаге индукции мы предполагаем, что требуемое свойство выполняется для всех поддеревьев (т. е. если $\Gamma \vdash s : S$ и $s \rightarrow s'$, то $\Gamma \vdash s' : S$, если $\Gamma \vdash s : S$ доказано подвыводом текущего дерева вывода). Затем мы рассматриваем варианты последнего правила в дереве.

Вариант T-VAR: $t = x \quad x:T \in \Gamma$.

Не может возникнуть (не существует правил вычисления, имеющих слева переменную).

Вариант T-Abs: $t = \lambda x:T_1. t_2$

Не может возникнуть.

Вариант T-APP: $t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}$

$T = T_{12}$

Рассматривая правила вычисления на рис. 9.1 с применениями слева, мы видим, что $t \rightarrow t'$ может быть вычислено тремя способами: E-APP1, E-APP2 и E-APPABS. Каждый из этих случаев мы рассмотрим по отдельности.

Подвариант E-APP1: $t_1 \rightarrow t'_1 \quad t' = t'_1 \ t_2$

Согласно предположениям для варианта T-APP, в исходном дереве вывода типа есть поддерево с заключением $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$. Применив к этому утверждению предположение индукции, получаем $\Gamma \vdash t'_1 : T_{11} \rightarrow T_{12}$. Сочетая этот результат с замечанием, что $\Gamma \vdash t_2 : T_{11}$ (также одно из предположений в варианте T-APP), получаем возможность применить правило T-APP, заключая, что $\Gamma \vdash t' : T$.

Подвариант E-APP2: $t_1 = v_1$ (т. е., t_1 является значением)

$$\begin{aligned} t_2 &\rightarrow t'_2 \\ t' &= v_1 \ t'_2 \end{aligned}$$

Аналогично.

Подвариант E-APPABS: $t_1 = \lambda x:T_{11}. t_{12}$
 $t_2 = v_2$
 $t' = [x \mapsto v_2]t_{12}$

С помощью леммы об инверсии мы можем разобрать дерево вывода типа для $\lambda x:T_{11}. t_{12}$ и получить $\Gamma, x:T_{11} \vdash t_{12} : T_{12}$. Исходя из этого и леммы о подстановке (§9.3.8), получаем $\Gamma \vdash t' : T_{12}$.

Варианты с булевскими константами и условными выражениями доказываются так же, как в §8.3.3.

9.3.10. РЕШЕНИЕ: Терм $(\lambda x:\text{Bool}. \lambda y:\text{Bool}. y)$ (`true true`) не имеет правильного типа, но редуцируется к правильно типизированному терму $(\lambda y:\text{Bool}. y)$.

9.4.1. РЕШЕНИЕ: T-TRUE и T-FALSE — правила введения. T-IF — правило устранения. T-ZERO и T-SUCC — правила введения. T-PRED и T-IsZERO — правила устранения. Вопрос о том, являются ли `succ` и `pred` конструкциями введения или устраниния, требует небольшого раздумья, поскольку можно считать, что они не только создают, но и *используют* числа. Здесь ключевое наблюдение состоит в том, что при встрече `pred` и `succ` образуют редекс; тоже касается и `iszero`.

11.2.1. РЕШЕНИЕ: $t_1 = (\lambda x:\text{Unit}. x) \text{ unit}$
 $t_{i+1} = (\lambda f:\text{Unit} \rightarrow \text{Unit}. f(f(\text{unit}))) (\lambda x:\text{Unit}. t_i)$

11.3.2. РЕШЕНИЕ:

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda _. T_1 . t_2 : T_1 \rightarrow T_2} \quad (\text{T-WILDCARD})$$

$$(\lambda _. T_{11} . t_{12}) v_2 \rightarrow t_{12} \quad (\text{E-WILDCARD})$$

Доказательство того, что эти правила выводятся из определения сокращения, выглядит в точности как доказательство теоремы 11.3.1.

11.4.1. РЕШЕНИЕ: Первое задание не представляет труда: если мы разворачиваем присыпывание типа с помощью правила $t \text{ as } T = (\lambda x:T. x) t$, то простая проверка показывает, что правила типизации, а также правила вычисления для этой конструкции прямо выводятся из правил для абстракции и применения функций.

Если же мы введем энергичную версию правила присыпывания, то придется применить более сложное правило удаления сахара, которое откладывает вычисление до тех пор, пока конструкция присыпывания не будет отброшена. Например, так:

$$t \text{ as } T \stackrel{\text{def}}{=} (\lambda x:\text{Unit} \rightarrow T. x \text{ unit}) (\lambda y:\text{Unit}. t) \text{ с новой переменной } y$$

Разумеется, использование именно типа `Unit` здесь несущественно: подошел бы любой тип.

Тонкость здесь заключается в том, что удаление сахара, будучи интуитивно правильным, *не обладает* в точности свойствами, которые требует теорема 11.3.1. Так получается потому, что результат перевода исчезает за *два* шага вычисления, тогда как высокоуровневое правило E-ASCRIBE срабатывает за один шаг. Это, однако, не должно нас удивлять: будем рассматривать удаление сахара как простую форму компиляции, и заметим, что в скомпилированном виде почти *любая* высокоуровневая конструкция требует нескольких шагов вычисления целевого языка на одну элементарную редукцию исходного языка. В этом случае требуется ослабить требования теоремы 11.3.1, и утверждать, что каждый шаг вычисления высокоуровневой конструкции должен соответствовать некоторой *последовательности* низкоуровневых шагов:

если $t \rightarrow_E t'$, то $e(t) \rightarrow_I^* e(t')$

Последняя тонкость состоит в том, что обратное направление — возможность всегда установить обратное соответствие между редукциями обессахаренного терма и редукциями исходного терма — требует некоторой осторожности при формулировке, поскольку устранение обессахаренной конструкции приписывания типа занимает два шага. После первого из них низкоуровневый терм *не соответствует* ни удалению сахара из исходного терма с высокуюровневым приписыванием типа, ни терму, в котором это приписывание отброшено. Верно, однако, что результат первой редукции обессахаренного терма всегда можно «завершить», проведя еще один шаг вычисления и получив результат удаления сахара для другого высокуюровневого терма. Формально говоря, если $e(t) \rightarrow_I s$, то $s \rightarrow^* e(t')$, где $t \rightarrow_E t'$.

11.5.1. РЕШЕНИЕ: Вот что требуется добавить:

```
let rec eval1 ctx t = match t with
  ...
  | TmLet(fi,x,v1,t2) when isval ctx v1 →
    termSubstTop v1 t2
  | TmLet(fi,x,t1,t2) →
    let t1' = eval1 ctx t1 in
    TmLet(fi, x, t1', t2)
  ...
let rec typeof ctx t = match t with
  ...
  | TmLet(fi,x,t1,t2) →
    let tyT1 = typeof ctx t1 in
    let ctx' = addbinding ctx x (VarBind(tyT1)) in
    (typeof ctx' t2)
```

11.5.2. РЕШЕНИЕ: Это определение работает не слишком хорошо. Во-первых, оно изменяет порядок вычисления: правила E-LETV и E-LET задают порядок вызова по значению, где t_1 в $\text{let } t_1 \text{ in } t_2$ необходимо редуцировать до значения перед тем, как подставить его вместо x и начать работать над t_2 . Кроме того, хотя корректность правила типизации T-LET при новом преобразовании сохраняется (это прямо следует из леммы о подстановке, 9.3.8), некорректность типизации термов *может потеряться*. Например, неверно типизированный терм

```
let x = unit(unit) in unit
```

переводится в корректно типизированный терм `unit`: поскольку x не встречается в теле `unit`, неверно типизированный терм `unit(unit)` просто исчезает.

11.8.2. РЕШЕНИЕ: Один из способов добавления типов к образцам для записей изображен на рис. A.1. Правило типизации для обобщенной конструкции `let`, T-LET, ссылается на отдельное отношение «тиปизации образцов», которое (с алгоритмической точки зрения) принимает на входе образец и тип, и, в случае успеха, выдает контекст, содержащий связывания для переменных образца. Правило T-LET при проверке типов в теле t_2 добавляет этот контекст к текущему контексту Γ . (Мы всюду предполагаем, что множества переменных, связываемых в различных полях образца для записи, не пересекаются.)

$\rightarrow \{\} \text{ let } p \quad (\text{тиปизированные})$

Расширяет 11.8

Правила типизации образцов

Новые правила типизации $\Gamma \vdash t : T$

$$\vdash x : T \Rightarrow x : T \quad (\text{P-VAR})$$

$$\begin{array}{c} \vdash t_1 : T_1 \quad \vdash p : T_1 \Rightarrow \Delta \\ \Gamma, \Delta \vdash t_2 : T_2 \end{array}$$

$$\frac{\text{для каждого } i, \quad \vdash p_i : T_i \Rightarrow \Delta_i \quad \vdash \text{let } p = t_1 \text{ in } t_2 : T_2}{\vdash \{l_i = p_i^{i \in 1..n}\} : \{k_j : T_j^{j \in 1..m}\} \Rightarrow \Delta_1, \dots, \Delta_n} \quad (\text{P-RCD})$$

$$(\text{T-LET})$$

Рис. A.1. Типизированные образцы для записей

Заметим, что при желании мы могли бы несколько улучшить правило типизации образцов, позволив образцам указывать меньше полей, чем содержится в значении, с которым будет проводиться сопоставление:

$$\frac{\{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\} \quad \forall i \in 1..n. \exists j \in 1..m. l_i = k_j \text{ и } \vdash p_i : T_j \Rightarrow \Delta_i}{\vdash \{l_i = p_i^{i \in 1..n}\} \in \{k_j : T_j^{j \in 1..m}\} \Rightarrow \Delta_1, \dots, \Delta_n} \quad (\text{P-RCD}')$$

Если мы примем это правило, то можем перестать считать форму проекции поля базовой конструкцией, поскольку теперь ее можно рассматривать как синтаксический сахар:

$$t.l \stackrel{\text{def}}{=} \text{let } \{l=x\}=t \text{ in } x$$

Лемма о сохранении для расширенной таким образом системы доказывается почти так же, как и для простого типизированного лямбда-исчисления. Единственное требуемое расширение — это лемма, которая связывает отношение типизации образцов с операцией сопоставления, которая производится при исполнении программы. Если имеется подстановка σ и контекст Δ с той же областью определения, что у σ , то $\Gamma \vdash \sigma \models \Delta$ означает, что для каждого $x \in \text{dom}(\sigma)$ выполняется $\Gamma \vdash \sigma(x) : \Delta(x)$.

ЛЕММА: Если $\Gamma \vdash t : T$ и $\vdash p : T \Rightarrow \Delta$, то $\text{match}(p, t) = \sigma$, где $\Gamma \vdash \sigma \models \Delta$.

Введенная нами дополнительная нотация требует небольшого обобщения стандартной леммы о подстановке (9.3.8):

ЛЕММА: Если $\Gamma, \Delta \vdash t : T$ и $\Gamma \vdash \sigma \models \Delta$, то $\Gamma \vdash \sigma t : T$.

Теперь аргументация в лемме о сохранении для правила C-LET может следовать той же схеме, что и раньше, используя новые леммы для случая C-LET.

11.9.1. РЕШЕНИЕ:

Bool	$\stackrel{\text{def}}{=}$	Unit+Unit
true	$\stackrel{\text{def}}{=}$	inl unit
false	$\stackrel{\text{def}}{=}$	inr unit
if t_0 then t_1 else t_2	$\stackrel{\text{def}}{=}$	case t_0 of inl $x_1 \Rightarrow t_1 \quad \quad \text{inr } x_2 \Rightarrow t_2$ с новыми переменными x_1 и x_2

11.11.1. РЕШЕНИЕ:

```

equal =
fix
  ( $\lambda eq: Nat \rightarrow Nat \rightarrow Bool.$ 
    $\lambda m: Nat. \lambda n: Nat.$ 
    if iszero m then iszero n
    else if iszero n then false
    else eq (pred m) (pred n));

▷ equal : Nat → Nat → Bool

plus = fix ( $\lambda p: Nat \rightarrow Nat \rightarrow Nat.$ 
             $\lambda m: Nat. \lambda n: Nat.$ 
             if iszero m then n else succ (p (pred m) n));

▷ plus : Nat → Nat → Nat

times = fix ( $\lambda t: Nat \rightarrow Nat \rightarrow Nat.$ 
               $\lambda m: Nat. \lambda n: Nat.$ 
               if iszero m then 0 else plus n (t (pred m) n));

▷ times : Nat → Nat → Nat

factorial = fix ( $\lambda f: Nat \rightarrow Nat.$ 
                   $\lambda m: Nat.$ 
                   if iszero m then 1 else times m (f (pred m)));

▷ factorial : Nat → Nat

factorial 5;

▷ 120 : Nat

```

11.11.2. РЕШЕНИЕ:

```

letrec plus : Nat → Nat → Nat =
   $\lambda m: Nat. \lambda n: Nat.$ 
   if iszero m then n else succ (plus (pred m) n) in
letrec times : Nat → Nat → Nat =
   $\lambda m: Nat. \lambda n: Nat.$ 
   if iszero m then 0 else plus n (times (pred m) n) in
letrec factorial : Nat → Nat =
   $\lambda m: Nat.$ 

```

```

        if iszero m then 1 else times m (factorial (pred m)) in
factorial 5;

▷ 120 : Nat

```

11.12.1. Решение: Сюрприз! На самом деле, теорема о продвижении *не* выполняется. Например, выражение `head[T] nil[T]` находится в тупике — никакое правило вычисления к нему не применимо, но оно не является значением. В полноценном языке программирования с такой ситуацией можно спрашивать, если заставить `head[T] nil[T]` порождать исключение, не вставая в тупик: мы рассматриваем исключения в главе 14.

11.12.2. Решение: Не от всех: если мы уберем аннотацию на `nil`, то потеряем теорему о единственности типов. С операционной точки зрения, когда программа проверки типов видит `nil`, она знает, что требуется присвоить ему тип `List T` для некоторого `T`, однако она не знает, какой `T` выбрать. Разумеется, более изощренные алгоритмы типизации, вроде того, который используется в компиляторе OCaml, умеют догадываться, какой тип нужен. Мы вернемся к этому вопросу в главе 22.

12.1.1. Решение: При рассмотрении применений. Допустим, мы пытаемся доказать, что терм $t_1 \ t_2$ нормализуем. Из предположения индукции нам известно, что нормализуемы t_1 и t_2 ; пусть их нормальные формы будут v_1 и v_2 . Из леммы об инверсии отношения типизации (9.3.1) нам известно, что v_1 имеет тип $T_{11} \rightarrow T_{12}$ для некоторых T_{11} и T_{12} . Следовательно, по лемме о канонических формах (9.3.4), v_1 обязан иметь вид $\lambda x:T_{11}.t_{12}$. Но сведение $t_1 \ t_2$ к $(\lambda x:T_{11}.t_{12}) \ v_2$ не дает нам нормальной формы, поскольку теперь мы можем применить правило E-APPABS, получая $[x \mapsto v_2]t_{12}$; для завершения доказательства требуется продемонстрировать, что этот терм нормализуем. Однако этого мы сделать не можем, поскольку в общем случае этот терм может оказаться больше, чем исходный терм $t_1 \ t_2$ (при подстановке возникает столько копий v_2 , сколько было вхождений переменной x в t_{12}).

12.1.7. Решение: Определение 12.1.2 расширяется двумя дополнительными пунктами:

- $R_{\text{Bool}}(t)$ тогда и только тогда, когда вычисление t завершается.
- $R_{T_1 \times T_2}(t)$ тогда и только тогда, когда вычисление t завершается, а также имеет место $R_{T_1}(t.1)$ и $R_{T_2}(t.2)$.

Доказательство леммы 12.1.4 также дополняется пунктом:

- Допустим, $T = T_1 \times T_2$ для некоторых T_1 и T_2 . Для направления «только если» (\Rightarrow) мы предполагаем, что $R_T(t)$ и должны показать, что $R_T(t')$, где $t \rightarrow t'$. Из определения $R_{T_1 \times T_2}$ мы знаем, что $R_{T_1}(t.1)$ и $R_{T_2}(t.2)$. Однако, правила вычисления E-PROJ1 и E-PROJ2 говорят нам, что $t.1 \rightarrow t'.1$ и $t.2 \rightarrow t'.2$, так что, по предположению индукции, имеет место $R_{T_1}(t'.1)$ и $R_{T_2}(t'.2)$. Отсюда, по определению $R_{T_1 \times T_2}$, получаем, что $R_{T_1 \times T_2}(t')$. Направление «если» (\Leftarrow) доказывается аналогично.

Наконец, нужно добавить несколько вариантов (по одному на каждое новое правило типизации) в доказательство леммы 12.1.5:

Вариант T-IF: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \quad \Gamma \vdash t_1 : \text{Bool}$
 $\Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T$
где $\Gamma = x_1:T_1, \dots, x_n:T_n$

Пусть $\sigma = [x_1 \mapsto v_1] \cdots [x_n \mapsto v_n]$. Согласно предположению индукции, имеем $R_{\text{Bool}}(\sigma t_1)$, $R_T(\sigma t_2)$ и $R_T(\sigma t_3)$. По лемме 12.1.3, σt_1 , σt_2 и σt_3 нормализуемы; обозначим их значения v_1 , v_2 и v_3 . Согласно лемме 12.1.4, имеем $R_{\text{Bool}}(v_1)$, $R_T(v_2)$, и $R_T(v_3)$. Кроме того, ясно, что терм σt сам по себе нормализуем.

Продолжаем по индукции по T .

- Если $T = A$ или $T = \text{Bool}$, то $R_T(\sigma t)$ немедленно следует из того, что σt нормализуем.
- Если $T = T_1 \rightarrow T_2$, то нам требуется показать, что $R_{T_2}(\sigma t s)$ для произвольного $s \in R_{T_1}$. Поэтому предположим, что $s \in R_{T_1}$. Тогда, по правилам вычисления для условных выражений, мы видим, что либо $\sigma t s \rightarrow^* v_2 s$, либо $\sigma t s \rightarrow^* v_3 s$, в зависимости от того, является ли v_1 значением `true` или `false`. Но мы знаем, что $R_{T_2}(v_2 s)$ и $R_{T_2}(v_3 s)$ (по определению $R_{T_1} \rightarrow T_2$) и, кроме того, что $R_{T_1} \rightarrow T_2(v_2)$ и $R_{T_1} \rightarrow T_2(v_3)$. По лемме 12.1.4 $R_{T_1} \rightarrow T_2(\sigma t s)$, что и требуется.

Вариант T-TRUE: $t = \text{true} \quad T = \text{Bool}$

Утверждение следует непосредственно.

Вариант T-FALSE: $t = \text{false} \quad T = \text{Bool}$

Утверждение следует непосредственно.

Вариант T-PAIR: $t = \{t_1, t_2\} \quad \Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T = T_1 \times T_2$

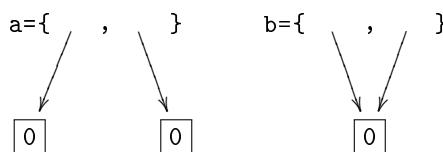
Согласно предположению индукции, $R_T(\sigma t_i)$ для $i = 1, 2$. Пусть нормальной формой для каждого σt_i будет v_i . Заметим, что по лемме 12.1.4, имеем $R_{T_i}(v_i)$.

Согласно правилам вычисления для первой и второй проекции $\{\sigma t_1, \sigma t_2\}.i \rightarrow^* v_i$, откуда по лемме 12.1.4 получаем, что $R_{T_i}(\{\sigma t_1, \sigma t_2\}.i)$. Из определения $R_{T_1 \times T_2}$ получаем, что $R_{T_1 \times T_2}(\{\sigma t_1, \sigma t_2\})$, то есть, $R_{T_1 \times T_2}(\sigma \{t_1, t_2\})$.

Вариант T-PROJ1: $t = t_0.1 \quad \Gamma \vdash t_0 : T_1 \times T_2 \quad T = T_1$

Утверждение непосредственно следует из определения $R_{T_1 \times T_2}$.

13.1.1. РЕШЕНИЕ:



13.1.2. РЕШЕНИЕ: Нет — вызовы `lookup` с любым индексом, кроме заданного в `update`, теперь будут зацикливаться. Дело в том, что обязательно нужно

искать значение, хранившееся в `a`, *до того*, как мы поместили в ячейку новую функцию. В противном случае, когда нам потребуется вызвать поиск, мы обнаружим в ячейке новую функцию, а не исходную.

13.1.3. Решение: Допустим, в языке имеется примитив `free`, который принимает в качестве аргумента ссылочную ячейку, и освобождает занимаемую ей память так, чтобы (скажем) следующая же операция выделения получила тот же самый участок памяти. Тогда программа

```
let r = ref 0 in
let s = r in
  free r;
let t = ref true in
  t := false;
succ (!s)
```

при вычислении приведет к тупиковому терму `succ false`. Заметим, что в возникновении проблемы ключевую роль играют псевдонимы: они не дают нам отслеживать ошибочные операции освобождения, просто запрещая `free` в ситуациях, когда переменная `r` присутствует в остатке вычисляемого выражения в качестве свободной.

Примеры такого рода легко получить в языках вроде C, где применяется ручное управление памятью (наш конструктор `ref` соответствует `malloc` в C, а наша операция `free` — упрощенная версия `free` из C).

13.3.1. Решение: Простое формальное описание сборки мусора может выглядеть так:

1. Конечность памяти можно смоделировать, считая множество адресов \mathcal{L} конечным.
2. Определим *достижимость* (reachability) адресов следующим образом. Обозначим множество адресов, упоминаемых в `t`, через $locations(t)$. Будем говорить, что адрес l' *достижим за один шаг* из адреса l при состоянии памяти μ , если $l' \in locations(\mu(l))$. Будем говорить, что l' *достижим* из l , если существует конечная последовательность адресов, начинаяющаяся с l и заканчивающаяся l' , так что каждый следующий адрес достижим из предыдущего за один шаг. Наконец, определим множество адресов, достижимых из терма `t` при состоянии памяти μ (записываеться в виде $reachable(t, \mu)$), как адреса, достижимые при состоянии μ из $locations(t)$.
3. Промоделируем действие сборки мусора как отношение $t \mid \mu \rightarrow_{gc} t \mid \mu'$, которое определяется следующим правилом:

$$\frac{\mu' = \mu, \text{ ограниченное областью } reachable(t, \mu)}{t \mid \mu \rightarrow_{gc} t \mid \mu'} \quad (\text{E-GC})$$

(т. е. область определения μ' — только $reachable(t, \mu)$, а его значение в каждой точке области определения совпадает со значением μ .)

4. Определим последовательности вычисления как последовательности обычных шагов вычисления, чередующихся с шагами сборки мусора: $\xrightarrow{gc}^* \stackrel{\text{def}}{=} (\rightarrow \cup \rightarrow_{gc})^*$. Обратите внимание, что мы не просто добавляем правило E-GC в обыкновенное одношаговое отношение вычисления: важно, что мы производим сборку мусора только на «внешнем уровне», где виден весь вычисляемый в данный момент терм. Если бы мы разрешили сборку мусора, например, «внутри» вычисления левой части терма-применения, мы бы могли по ошибке собрать и заново использовать адреса, которые упоминаются в правой части, поскольку, глядя на левую часть, мы не заметили бы, что они все еще доступны.

5. Докажем корректность нашего определения вычисления, показав, что оно не изменяет окончательные результаты, за исключением возможности исчерпания памяти:

- (a) Если $t \mid \mu \xrightarrow{gc}^* t' \mid \mu''$, то $t \mid \mu \rightarrow^* t' \mid \mu'$ для некоторого μ' , такого, что μ' имеет более широкую область определения, чем μ'' , и там, где они оба определены, они совпадают.
- (b) Если $t \mid \mu \rightarrow^* t' \mid \mu'$ то
 - i. либо $t \mid \mu \xrightarrow{gc}^* t' \mid \mu''$ для некоторого μ'' , имеющего более узкую область определения, чем μ' , и там, где они оба определены, они совпадают;
 - ii. либо вычисление $t \mid \mu$ исчерпывает память, т. е. достигает такого состояния $t''' \mid \mu'''$, что следующий шаг вычисления t''' требует выделения памяти, но это невозможно, так как $reachable(t''', \mu''') = \mathcal{L}$.

Этот простой подход к описанию сборки мусора не учитывает некоторые свойства реальной памяти, например, то, что хранение различных видов значений, как правило, требует различного количества памяти. Игнорируются также некоторые сложные языковые конструкции, например, *финализаторы* (finalizers, фрагменты, которые исполняются, когда система исполнения собирается освободить структуры данных, к которым они приписаны) и *слабые указатели* (weak pointers, указатели, которые не считаются «настоящими» ссылками на структуру данных, так что эта структура может быть освобождена, даже если на нее имеются слабые указатели). Более тщательное обсуждение сборки мусора средствами операционной семантики можно найти у Morrisett и др. (Morrisett, Felleisen, and Harper, 1995).

13.4.1. РЕШЕНИЕ:

```
let r1 = ref (λx:Nat.0) in
let r2 = ref (λx:Nat.(!r1)x) in
(r1 := (λx:Nat.(!r2)x);
 r2);
```

13.5.2. РЕШЕНИЕ: Пусть μ — состояние памяти с единственным адресом 1

$$\mu = (1 \mapsto \lambda x: \text{Unit}. (!1)(x))$$

а Γ — пустой контекст. Тогда μ правильно типизировано по отношению к таким двум структурам типизации памяти:

$$\begin{aligned}\Sigma_1 &= 1: \text{Unit} \rightarrow \text{Unit} \\ \Sigma_2 &= 1: \text{Unit} \rightarrow (\text{Unit} \rightarrow \text{Unit})\end{aligned}$$

13.5.8. РЕШЕНИЕ: В этой системе есть правильно типизированные термы, не являющиеся сильно нормализуемыми. Один пример был приведен в упражнении 13.1.2. Вот еще один:

```
t1 = λr:Ref (Unit → Unit)
            (r := (λx:Unit. (!r)x);
             (!r) unit);
t2 = ref (λx:Unit. x);
```

Применение $t1$ к $t2$ дает (правильно типизированный) расходящийся терм.

В общем случае можно определять произвольные рекурсивные функции с помощью ссылок. (Такой метод действительно используется в некоторых реализациях функциональных языков.)

- Выделим ссылочную ячейку и инициализируем ее функцией-заглушкой нужного нам типа:

```
factref = ref (λn:Nat. 0);
▷ factref : Ref (Nat → Nat)
```

- Определим тело нужной функции, используя для рекурсивных вызовов обращение к содержимому ячейки:

```
factbody =
  λn:Nat.
    if iszero n then 1 else times n ((!factref)(pred n));
▷ factbody : Nat → Nat
```

- «Вернем тело на место», в ссылочную ячейку:

```
factref := factbody;
```

- Обратимся к содержимому ячейки и используем его:

```
fact = !factref;
fact 5;
▷ 120 : Nat
```

14.1.1. РЕШЕНИЕ: Указание ожидаемого типа `error` уничтожит свойство сохранения типов. Например, правильно типизированный терм

$$(\lambda x:\text{Nat}.x) \ ((\lambda y:\text{Bool}.5) \ (\text{error as Bool}));$$

(где `error as T` — синтаксис для указания типа в исключениях) за один шаг превратится в неверно типизированный терм

$$(\lambda x:\text{Nat}.x) \ (\text{error as Bool});$$

Когда правила вычисления распространяют `error` от места возникновения до верхних уровней программы, нам приходится рассматривать его как терм, имеющий различные типы. Гибкость правила T-ERROR нам это позволяет.

14.3.1. РЕШЕНИЕ: В «Определении Standard ML» (Milner, Tofte, Harper, and MacQueen, 1997; Milner and Tofte, 1991b) формально описывается тип `exn`. Подобный подход можно также найти у Харпера и Стоуна (Harper and Stone, 2000).

14.3.2. РЕШЕНИЕ: См. Леруа и Пессо (Leroy and Pessaux, 2000).

14.3.3. РЕШЕНИЕ: См. Харпер и др. (Harper, Duba, and MacQueen, 1993).

15.2.1. РЕШЕНИЕ:

$$\frac{\begin{array}{c} \{x:\text{Nat},y:\text{Nat},z:\text{Nat}\} \\ \langle : \{y:\text{Nat},x:\text{Nat},z:\text{Nat}\} \end{array}}{\{x:\text{Nat},y:\text{Nat},z:\text{Nat}\} \langle : \{y:\text{Nat}\}} \quad \frac{\begin{array}{c} \{y:\text{Nat},x:\text{Nat},z:\text{Nat}\} \\ \langle : \{y:\text{Nat}\} \end{array}}{\{x:\text{Nat},y:\text{Nat},z:\text{Nat}\} \langle : \{y:\text{Nat}\}} \quad \frac{}{\{x:\text{Nat},y:\text{Nat},z:\text{Nat}\} \langle : \{y:\text{Nat}\}}$$

S-RCDPERM S-RCDWIDTH S-TRANS

15.2.2. РЕШЕНИЕ: Существует множество других деревьев вывода с тем же самым заключением. Вот одно из них:

$$\frac{\frac{\frac{\vdots}{Rxy \langle : Rx} \text{S-RCDWIDTH} \quad \frac{\vdots}{\text{Nat} \langle : \text{Nat}} \text{S-REFL}}{\text{Rx} \rightarrow \text{Nat} \langle : Rxy \rightarrow \text{Nat}} \text{S-ARROW}}{\vdash f : Rx \rightarrow \text{Nat}} \text{T-SUB} \quad \frac{\vdots}{\vdash xy : Rxy} \text{T-APP}$$

$\vdash f : Rxy \rightarrow \text{Nat}$

Вот еще одно:

$$\frac{\frac{\frac{\vdots}{T-RCD}}{\vdash xy : Rxy} \text{T-RCD} \quad \frac{\frac{\vdots}{Rxy \langle : Rx} \text{S-RCDWIDTH} \quad \frac{\vdots}{Rx \langle : Rx} \text{S-REFL}}{Rxy \langle : Rx} \text{S-TRANS}}{\vdash xy : Rx} \text{T-SUB}$$

$\vdash f : Rx \rightarrow \text{Nat}$

T-A

В сущности, как можно догадаться по второму примеру, в этом исчислении имеется бесконечно много деревьев вывода для любого выводимого утверждения.

15.2.3. РЕШЕНИЕ:

1. Я насчитал шесть: $\{a:\text{Top}, b:\text{Top}\}$, $\{b:\text{Top}, a:\text{Top}\}$, $\{a:\text{Top}\}$, $\{b:\text{Top}\}$, $\{\}$ и Top .

2. Пусть, например,

$$\begin{aligned} S_0 &= \{\} \\ S_1 &= \{a:\text{Top}\} \\ S_2 &= \{a:\text{Top}, b:\text{Top}\} \\ S_3 &= \{a:\text{Top}, b:\text{Top}, c:\text{Top}\} \\ &\text{и т. д.} \end{aligned}$$

3. Пусть, например, $T_0 = S_0 \rightarrow \text{Top}$, $T_1 = S_1 \rightarrow \text{Top}$, $T_2 = S_2 \rightarrow \text{Top}$, и т. д.

15.2.4. Решение: (1) Нет. Если бы такой тип существовал, это был бы либо функциональный тип, либо тип записей (очевидно, это не может быть Top); однако, тип записей не мог бы быть подтипов никакого функционального типа, и наоборот. (2) Тоже нет. Если бы существовал такой функциональный тип $T_1 \rightarrow T_2$, то тип его области определения T_1 был бы подтипов любого другого типа S_1 , а это, как мы только что убедились, невозможно.

15.2.5. Решение: Добавление этого правила было бы неверным шагом, если мы желаем сохранить имеющуюся семантику вычисления. Новое правило позволило бы нам вывести, например, что $\text{Nat} \times \text{Bool} <: \text{Nat}$. Отсюда получалось бы, что тупиковый терм ($\text{succ}(5, \text{true})$) правильно типизирован, а это нарушило бы теорему о продвижении. Такое правило безопасно в семантике на основе преобразования типов (см. 15.6), но даже там возникают некоторые алгоритмические сложности при проверке подтиповирования.

15.3.1. Решение: Добавляя в отношение подтиповирования лишние пары, мы можем потерять как сохранение, так и продвижение. Например, добавление аксиомы

$$\{x:\{\}\} <: \{x:\text{Top} \rightarrow \text{Top}\}$$

позволяет вывести, что $\Gamma \vdash t : \text{Top}$, где $t = (\{x=\{\}\}.x)\{\}$. Но $t \rightarrow \{\}\{\}$, который не типизируется — нарушается свойство сохранения. С другой стороны, если мы добавим аксиому

$$\{\} <: \text{Top} \rightarrow \text{Top}$$

то терм $\{\}\{\}$, наоборот, будет типизируем, но это тупиковый терм, не являющийся значением — нарушается свойство продвижения.

Напротив, исключение пар из отношения подтиповирования никакого вреда не причиняет. В формулировке теоремы о продвижении отношение типизации упоминается лишь в предпосылках, так что ограничение отношения подтиповирования, влекущее ограничение отношения типизации, может только упростить задачу доказательства этой теоремы. В случае теоремы о сохранении мы можем ожидать, в частности, что потеря свойства транзитивности создаст проблемы. Однако, таких проблем не возникает; с интуитивной точки зрения это потому, что правило транзитивности не играет существенной роли в системе: транзитивность можно восстановить при помощи правила включения

T-SUB. Например, вместо

$$\frac{\frac{\vdots \quad \vdots}{\Gamma \vdash t \in S} \quad \frac{S <: U \quad U <: T}{S <: T} S\text{-TRANS}}{\Gamma \vdash t : T} T\text{-SUB}$$

всегда можно написать

$$\frac{\frac{\vdots \quad \vdots}{\Gamma \vdash t : S} \quad \frac{\Gamma \vdash t \in U \quad S <: U}{U <: T} T\text{-SUB}}{\Gamma \vdash t : T} T\text{-SUB}$$

15.3.2. РЕШЕНИЕ:

- Индукция по деревьям вывода подтипования. Рассмотрев правила подтипования на рис. 15.1 и 15.3, мы видим, что финальным правилом при выводе $S <: T_1 \rightarrow T_2$ должно быть S-REFL, S-TRANS или S-ARROW. Если последнее правило — S-REFL, то результат следует немедленно (поскольку в этом случае $S = T_1 \rightarrow T_2$, а из рефлексивности мы можем вывести $T_1 <: T_1$ и $T_2 <: T_2$).

Если последнее правило — S-TRANS, то у нас имеются подвыводы с заключениями $S <: U$ и $U <: T_1 \rightarrow T_2$ для некоторого типа U . Применяя предположение индукции ко второму подвыводу, мы видим, что U обязан иметь вид $U_1 \rightarrow U_2$, причем $T_1 <: U_1$ и $U_2 <: T_2$. Поскольку U — функциональный тип, мы можем применить предположение индукции к первому подвыводу, и получить $S = S_1 \rightarrow S_2$, причем $U_1 <: S_1$ и $S_2 <: U_2$. Наконец, применяя дважды правило S-TRANS, мы можем собрать вместе полученные нами утверждения и получить $T_1 <: S_1$ (из $T_1 <: U_1$ и $U_1 <: S_1$) и $S_2 <: T_2$ (из $S_2 <: U_2$ и $U_2 <: T_2$).

Если последнее правило — S-ARROW, то S имеет требуемый вид, а заключения непосредственных подвыводов есть в точности те утверждения, которые нам требуется получить о частях S .

- Индукция по деревьям вывода. Снова рассматривая правила подтипования, убеждаемся, что финальным правилом в выводе $S <: \{l_i : T_i\}_{i \in 1..n}$ может быть S-REFL, S-TRANS, S-RCDWIDTH, S-RCDDEPTH либо S-RCDPERM. Вариант S-REFL тривиален. В вариантах S-RCDWIDTH, S-RCDDEPTH и S-RCDPERM требуемое утверждение следует непосредственно.

Если последнее правило — S-TRANS, то у нас должны быть подвыводы с заключениями $S <: U$ и $U <: \{l_i : T_i\}_{i \in 1..n}$ для некоторого типа U .

Применяя предположение индукции ко второму подвыходу, мы видим, что U имеет вид $\{u_a : U_a^{a \in 1..o}\}$, причем $\{l_i^{i \in 1..n}\} \subseteq \{u_a^{a \in 1..o}\}$ и $U_a <: T_i$ для каждого $l_i = u_a$. Поскольку теперь мы знаем, что U — тип записей, мы можем применить предположение индукции к первому подвыходу и получить $S = \{k_j : S_j^{j \in 1..m}\}$, причем $\{u_a^{a \in 1..o}\} \subseteq \{k_j^{j \in 1..m}\}$ и $S_j <: U_a$ для каждого $u_a = k_j$. Собирая вместе эти утверждения, получаем, что $\{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\}$ по транзитивности отношения «подмножество», и $S_j <: T_i$ по $S\text{-TRANS}$ для каждого $l_i = k_j$, поскольку каждая метка в типе T должна также присутствовать в U (т. е. l_i должно равняться u_a для некоторого a), и тогда нам известно, что $S_j <: U_a$ и $U_a <: T_i$. (Странные имена метапеременных в этом доказательстве неизбежны: иначе просто не хватает латинских букв.)

15.3.6. Решение: Обе части доказываются индукцией по деревьям вывода типов. Мы приводим доказательство только первой части.

Рассматривая правила типизации, мы видим, что финальным правилом при выводе $\vdash v : T_1 \rightarrow T_2$ может быть только $T\text{-ABS}$ или $T\text{-SUB}$. Если это $T\text{-ABS}$, то необходимый нам результат непосредственно следует из предпосылки правила. Предположим тогда, что последнее правило — $T\text{-SUB}$.

Из предпосылок $T\text{-SUB}$ имеем $\vdash v : S$ и $S <: T_1 \rightarrow T_2$. По лемме об инверсии (15.3.2) S имеет вид $S_1 \rightarrow S_2$. Тогда результат следует из предположения индукции.

16.1.2. Решение: Часть (1) доказывается прямолинейной индукцией по структуре S .

Для доказательства части (2) заметим сначала, что если существует какой-либо вывод $S <: T$, то согласно части (1), существует вывод без использования рефлексивности. Мы можем доказать наше утверждение индукцией по размеру дерева вывода $S <: T$, не использующего рефлексивность. Заметим, что мы строим индукцию по *размеру* дерева, а не по его структуре, как раньше. Это необходимо потому, что в случаях функциональных типов и типов-записей мы применяем индуктивное предположение к *новым построенным* деревьям вывода, которые не являются поддеревьями исходного вывода.

Если последнее правило вывода не является экземпляром $S\text{-TRANS}$, то результат прямо следует из индуктивного предположения (т. е. по предположению индукции все подвыходы последнего правила можно заменить выводами без использования транзитивности; поскольку последнее правило также не использует транзитивность, весь вывод также не использует транзитивность). Предположим тогда, что последнее правило в выводе — $S\text{-TRANS}$, т. е. что у нас есть подвыходы с заключениями $S <: U$ и $U <: T$, для некоторого U . Рассмотрим варианты пар последних правил в обоих подвыходах.

Вариант Что угодно/S-Top: $T = \text{Top}$

Если правое поддерево заканчивается на $S\text{-TOP}$, то результат следует немедленно, поскольку $S <: \text{Top}$ можно вывести из $S\text{-TOP}$ независимо от формы S .

Вариант S-Top/Что угодно: $U = \text{Top}$

Если левое поддерево заканчивается на $S\text{-TOP}$, мы замечаем, что, согласно предположению индукции, можно предположить, что правое поддерево не

использует транзитивность. Рассмотрев правила подтиpирования, мы видим, что последним правилом в этом поддереве всегда будет S-TOP (мы уже исключили случай рефлексивности, а все остальные правила требуют, чтобы U был либо функциональным типом, либо типом записей). Нужный результат следует по S-TOP.

Вариант S-ARROW/S-ARROW:

$$\begin{array}{lll} S = S_1 \rightarrow S_2 & U = U_1 \rightarrow U_2 & T = T_1 \rightarrow T_2 \\ U_1 <: S_1 & S_2 <: U_2 \\ T_1 <: U_1 & U_2 <: T_2 \end{array}$$

Используя S-TRANS, можно построить выводы $T_1 <: S_1$ и $S_2 <: T_2$, исходя из данных подвыводов. Более того, эти выводы строго короче исходного вывода, так что мы можем применить предположение индукции и получить выводы $T_1 <: S_1$ и $S_2 <: T_2$ без использования транзитивности. Сочетая их с правилом S-ARROW, получаем вывод $S_1 \rightarrow S_2 <: T_1 \rightarrow T_2$ без транзитивности.

Вариант S-RCD/S-RCD:

Аналогично.

Остальные варианты:

Остальные сочетания (S-ARROW/S-RCD и S-RCD/S-ARROW) возникнуть не могут, поскольку при этом на форму U накладываются противоречивые требования.

16.1.3. РЕШЕНИЕ: При добавлении типа `Bool` первая часть леммы 16.1.2 потребует небольшой модификации. Она должна гласить: « $S <: S$ можно вывести без использования S-REFL для любого типа S , кроме `Bool`». Второй вариант — добавить правило

$$\text{Bool} <: \text{Bool}$$

в определение отношения подтиpирования, а затем показать, что S-REFL можно исключить из расширенной таким образом системы.

16.2.5. РЕШЕНИЕ: Индукция по деревьям декларативного вывода типов. Рассмотрим варианты финального правила в дереве.

Вариант T-VAR: $t = x \quad \Gamma(x) = T$

Утверждение следует непосредственно, по правилу TA-VAR.

Вариант T-Abs: $t = \lambda x:T_1 . t_2 \quad \Gamma, x:T_1 \vdash t_2 : T_2 \quad T = T_1 \rightarrow T_2$

По предположению индукции, $\Gamma, x:T_1 \vdash t_2 : S_2$ для некоторого $S_2 <: T_2$. Согласно TA-Abs, $\Gamma \vdash t : T_1 \rightarrow S_2$. По правилу S-ARROW, $T_1 \rightarrow S_2 <: T_1 \rightarrow T_2$, что нам и требуется.

Вариант T-APP: $t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$

По предположению индукции, $\Gamma \vdash t_1 : S_1$ для некоторого $S_1 <: T_{11} \rightarrow T_{12}$, и $\Gamma \vdash t_2 : S_2$ для некоторого $S_2 <: T_{11}$. Согласно лемме об инверсии отношения подтиpирования (15.3.2), тип S_1 должен иметь вид $S_{11} \rightarrow S_{12}$ для некоторых S_{11} и S_{12} , причем $T_{11} <: S_{11}$ и $S_{12} <: T_{12}$. По транзитивности, $S_2 <: S_{11}$. Из полноты алгоритмического отношения подтиpирования следует, что $\vdash S_2 <: S_{11}$. Теперь применим TA-APP и получим $\Gamma \vdash t_1 \ t_2 : S_{12}$, что завершает рассмотрение данного варианта (поскольку мы уже имеем $S_{12} <: T_{12}$).

Вариант T-RCD: $t = \{l_i=t_i^{i \in 1..n}\}$ $\Gamma \vdash t_i : T_i$ для каждого i
 $T = \{l_i:T_i^{i \in 1..n}\}$

Не представляет труда.

Вариант T-PROJ: $t = t_1.l_j$ $\Gamma \vdash t_1 : \{l_i:T_i^{i \in 1..n}\}$ $T = T_j$
Доказательство аналогично варианту с термом-применением.

Вариант T-SUB: $t : S$ $S <: T$

Утверждение следует из предположения индукции и транзитивности подтиповирования.

16.2.6. РЕШЕНИЕ: Терм $\lambda x:\{a:Nat\}.x$ имеет, согласно декларативным правилам, типы $\{a:Nat\} \rightarrow \{a:Nat\}$ и $\{a:Nat\} \rightarrow \text{Top}$. Однако без правила S-ARROW эти типы оказываются несравнимы (и не имеется типа, лежащего ниже их обоих).

16.3.2. РЕШЕНИЕ: Прежде всего, приведем пару взаимно рекурсивных алгоритмов, которые, как мы утверждаем (и намереваемся доказать), вычисляют, соответственно, объединение J и пересечение M пары типов S и T. Второй из этих алгоритмов также может вернуть значение *неудача*, что означает отсутствие у S и T пересечения.

$$S \vee T = \begin{cases} \text{Bool} & \text{если } S = T = \text{Bool} \\ M_1 \rightarrow J_2 & \text{если } S = S_1 \rightarrow S_2 \quad T = T_1 \rightarrow T_2 \\ \{j_l:J_l^{l \in 1..q}\} & \text{если } S = \{k_j:S_j^{j \in 1..m}\} \\ & T = \{l_i:T_i^{i \in 1..n}\} \\ & \{j_l^{l \in 1..q}\} = \{k_j^{j \in 1..m}\} \cap \{l_i^{i \in 1..n}\} \\ & S_j \vee T_i = J_l \quad \text{для каждого } j_l = k_j = l_i \\ \text{Top} & \text{в остальных случаях} \end{cases}$$

$$S \wedge T = \begin{cases} S & \text{если } T = \text{Top} \\ T & \text{если } S = \text{Top} \\ \text{Bool} & \text{если } S = T = \text{Bool} \\ J_1 \rightarrow M_2 & \text{если } S = S_1 \rightarrow S_2 \quad T = T_1 \rightarrow T_2 \\ \{m_l:M_l^{l \in 1..q}\} & \text{если } S = \{k_j:S_j^{j \in 1..m}\} \\ & T = \{l_i:T_i^{i \in 1..n}\} \\ & \{m_l^{l \in 1..q}\} = \{k_j^{j \in 1..m}\} \cup \{l_i^{i \in 1..n}\} \\ & S_j \wedge T_i = M_l \quad \text{для каждого } m_l = k_j = l_i \\ & M_l = S_j \quad \text{если } m_l = k_j \text{ встречается только в } S \\ & M_l = T_i \quad \text{если } m_l = l_i \text{ встречается только в } T \\ \text{неудача} & \text{в остальных случаях} \end{cases}$$

В случае функционального типа в первом алгоритме вызов второго алгоритма может привести к неудаче; в таком случае, первый алгоритм проходит до последнего варианта и получает результат Top. Например, $\text{Bool} \rightarrow \text{Top} \vee \{\} \rightarrow \text{Top} = \text{Top}$.

Несложно убедиться в том, что функции \vee и \wedge всюду определены (т. е. их вычисление всегда завершается): заметим только, что общий размер S и T становится меньше при каждом рекурсивном вызове. Кроме того, алгоритм \wedge никогда не завершается неудачей, если его аргументы ограничены снизу.

Лемма A.10 Если $L <: S \cup L <: T$, то $S \wedge T = M$ для некоторого M .

Доказательство: Индукция по размеру S (или, что равносильно, T) с разбором вариантов формы S и T . Если какой-либо из типов S или T равен `Top`, то применим один из двух первых вариантов определения, и результатом будет, соответственно, T или S . Не может быть, чтобы S и T имели разную форму, поскольку лемма об инверсии отношения подтипования (15.3.2) предъявляла бы тогда противоречивые требования к L ; например, если S — функциональный тип, то таковым должен быть и L , но если T — тип записей, то L также должен оказаться типом записей.¹ Остается три варианта.

Если S и T оба равны `Bool`, то применим третий вариант определения, и задача решена.

Предположим тогда, что $S = S_1 \rightarrow S_2$ и $T = T_1 \rightarrow T_2$. Из полноты алгоритма для \vee следует, что первый рекурсивный вызов имеет результатом некоторый тип J_1 . При этом по лемме об инверсии L обязан иметь вид $L_1 \rightarrow L_2$, причем $L_2 <: S_2$ и $L_2 <: T_2$. Значит, L_2 является общей нижней гранью S_2 и T_2 ; таким образом, применимо предположение индукции, и мы знаем, что $S_2 \wedge L_2$ не завершается неудачей, а возвращает некий тип M_2 . Таким образом, $S \wedge T = J_1 \rightarrow M_2$.

Наконец, предположим, что $S = \{k_j : S_j\}_{j \in 1..m}$, а $T = \{l_i : T_i\}_{i \in 1..n}$. По лемме об инверсии L обязан быть типом записей, и его метки должны включать все метки, встречающиеся в S либо T . Более того, для каждой метки, имеющейся как в S , так и в T , согласно лемме об инверсии соответствующее поле в L будет общим подтипом S и T . Это гарантирует нам, что рекурсивные вызовы алгоритма \wedge для всех общих меток полей будут успешны.

Теперь нужно убедиться, что эти алгоритмы вычисляют объединения и пересечения. Доказательство разбито на два утверждения: утверждение A.11 показывает, что вычисленное пересечение является общей нижней гранью S и T , а объединение является верхней гранью; утверждение A.12 показывает, что вычисленное пересечение больше любой общей нижней грани S и T , а вычисленное объединение меньше любой общей верхней грани.

Утверждение A.11

1. Если $S \vee T = J$, то $S <: J \cup T <: J$.
2. Если $S \wedge T = M$, то $M <: S \cup M <: T$.

Доказательство: Прямолинейная индукция по размеру «вывода» $S \wedge T = M$ или $S \vee T = J$ (т. е. по количеству рекурсивных вызовов определений \wedge и \vee , требуемых для вычисления M либо J).

Утверждение A.12

1. Предположим, что $S \vee T = J$ и, для некоторого U , $S <: U \cup T <: U$. Тогда $J <: U$.

¹Строго говоря, в лемме 15.3.2 не упоминается `Bool`. Дополнительный вариант для этого случая просто говорит, что единственным подтипом `Bool` является он сам.

2. Предположим, что $S \wedge T = M$ и, для некоторого L , $L <: S \wedge L <: T$. Тогда $L <: M$.

Доказательство: Две части леммы доказываются совместно, индукцией по размеру S и T (в сущности, можно использовать индукцию почти по чему угодно). Для данных S и T рассмотрим обе части по очереди.

В части (1) рассмотрим варианты формы для U . Если U равняется Top , ничего доказывать не надо, поскольку $J <: \text{Top}$ при любом J . Если $U = \text{Bool}$, то, согласно лемме об инверсии (15.3.2), S и T также должны равняться Bool , так что $J = \text{Bool}$, и доказательство закончено. Остальные варианты более интересны.

Если $U = U_1 \rightarrow U_2$, то, согласно лемме об инверсии, $S = S_1 \rightarrow S_2$ и $T = T_1 \rightarrow T_2$, причем $U_1 <: S_1$, $U_1 <: T_1$, $S_2 <: U_2$ и $T_2 <: U_2$. Согласно предположению индукции M_1 (пересечение S_1 и T_1) лежит выше U_1 , тогда как J_2 (объединение S_2 и T_2) лежит ниже, чем U_2 . По правилу S-ARROW, $M_1 \rightarrow J_2 <: U_1 \rightarrow U_2$.

Если U — тип записей, то, согласно лемме об инверсии, S и T также являются типами записей. Более того, множества меток S и T — надмножества множества меток U , и тип каждого поля U является надтиповом типов соответствующих меток в S и T . Таким образом, объединение S и T будет содержать, по меньшей мере, метки U , и (по предположению индукции) поля объединения будут подтипами соответствующих полей U . По правилу S-RCD, $J <: U$.

В части (2) мы также рассматриваем варианты формы для S и T . Если один из этих типов — Top , то пересечение равно второму типу, и результат следует непосредственно. Варианты, в которых S и T имеют разные (не- Top) формы, возникнуть не могут, как мы уже видели в доказательстве A.10. Если оба типа равны Bool , то результат, опять же, следует непосредственно. Оставшиеся случаи (S и T — либо оба функциональные типы, либо оба типы записей) более интересны.

Если $S = S_1 \rightarrow S_2$ и $T = T_1 \rightarrow T_2$, то по лемме об инверсии имеем $L = L_1 \rightarrow L_2$, причем $S_1 <: L_1$, $T_1 <: L_1$, $L_2 <: S_2$ и $L_2 <: T_2$. По предположению индукции J_1 (объединение S_1 и T_1) лежит ниже L_1 , а M_2 (пересечение S_2 и T_2) лежит выше L_2 . По правилу S-ARROW, $L_1 \rightarrow L_2 <: J_1 \rightarrow M_2$.

Если S и T — типы записей, то по лемме об инверсии L также является типом записей. Более того, L должен содержать все метки полей, имеющиеся в S или T (и, возможно, какие-то еще), а соответствующие поля должны находиться в отношении подтипования. Для каждого поля m_l в M (пересечении S и T) возможны три варианта. Если m_l встречается как в S , так и в T , то его тип в M есть пересечение его типов в S и T , а соответствующий тип в L является подтиповом поля в M по предположению

индукции. Если же t_1 встречается только в S , то соответствующий тип в M тот же, что и в S , и мы знаем, что тип в L меньше него. Так же можно рассуждать и в том случае, когда t_1 встречается только в T .

16.3.3. РЕШЕНИЕ: Минимальный тип этого терма **Top** — объединение **Bool** и $\{\}$. Однако тот факт, что этот терм типизируем, вероятно, должно рассматриваться как слабость нашего языка, поскольку трудно себе представить, чтобы программист действительно хотел написать это выражение — в конце концов, к значению типа **Top** нельзя применить никакую операцию, так что нет никакого смысла его вычислять! Справиться с этим недостатком можно двумя способами. Во-первых, можно просто исключить **Top** из системы, и сделать \vee частичной операцией. Во-вторых, можно сохранить **Top**, но заставить программу проверки типов выдавать предупреждение, когда обнаруживается терм с минимальным типом **Top**.

16.3.4. РЕШЕНИЕ: Обработка типов **Ref** не представляет сложности. Нужно просто добавить по варианту в алгоритмы, вычисляющие объединение и пересечение:

$$S \vee T = \begin{cases} \dots & \dots \\ Ref(T_1) & \text{если } S = Ref(S_1), T = Ref(T_1), S_1 <: T_1 \text{ и } T_1 <: S_1 \\ \dots & \dots \end{cases}$$

$$S \wedge T = \begin{cases} \dots & \dots \\ Ref(T_1) & \text{если } S = Ref(S_1), T = Ref(T_1), S_1 <: T_1 \text{ и } T_1 <: S_1 \\ \dots & \dots \end{cases}$$

Однако, если мы разбиваем **Ref** на конструкторы **Source** и **Sink**, то возникает большая проблема: отношение подтипования больше не обладает объединениями (и пересечениями)! Например, типы **Ref{a:Nat,b:Bool}** и **Ref{a:Nat}** являются подтипами как **Source{a:Nat}**, так и **Sink{a:Nat,b:Bool}**, а у этих двух типов нет наибольшей нижней грани.

Справиться с этим затруднением можно несколькими способами. Возможно, самый простой из них — добавить в систему *либо* **Source**, либо **Sink**, но не оба вместе. Для многих областей применения этого достаточно. Например, для уточненной реализации классов в §18.12 нам требуется только **Source**. В параллельном языке с типами каналов (§15.5), наоборот, выгоднее использовать только **Sink**, поскольку при этом мы сможем определить процесс-сервер и передавать между процессами только «возможность посылки данных» в его канал доступа (возможность получения данных требуется только самому процессу-серверу).

Если имеются только типы **Source**, алгоритм вычисления объединений остается полным, если его расширить следующим образом (нужны также вышеприведенные варианты для **Ref**; аналогичные варианты нужно добавить в алгоритм вычисления пересечений):

$$S \vee T = \left\{ \begin{array}{ll} \dots & \dots \\ \text{Source}(J_1) & \text{если } S = \text{Ref}(S_1), T = \text{Ref}(T_1), S_1 \vee T_1 = J_1 \\ \text{Source}(J_1) & \text{если } S = \text{Source}(S_1), T = \text{Source}(T_1), S_1 \vee T_1 = J_1 \\ \text{Source}(J_1) & \text{если } S = \text{Ref}(S_1), T = \text{Source}(T_1), S_1 \vee T_1 = J_1 \\ \text{Source}(J_1) & \text{если } S = \text{Source}(S_1), T = \text{Ref}(T_1), S_1 \vee T_1 = J_1 \\ \dots & \dots \end{array} \right.$$

Еще одно решение (предложенное Хеннеси и Рили, Hennessy and Riely, 1998) состоит в том, чтобы модифицировать конструктор `Ref` так, чтобы он вместо одного принимал два аргумента: элементы типа `Ref S T` — ссылочные ячейки, в которые можно *записывать* элементы типа `S`, и из которых можно *считывать* элементы типа `T`. Новый конструктор `Ref` контравариантен по первому параметру и ковариантен по второму. Теперь можно определить `Sink S` как сокращение для `Ref S Top`, а `Source T` — как `Ref Bot T`.

16.4.1. РЕШЕНИЕ: Да:

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = \text{Bot} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3 \quad T_2 \vee T_3 = T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{TA-IF})$$

Альтернатива,

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = \text{Bot} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{Bot}} \quad (\text{TA-IFD})$$

кажется привлекательной, и была бы безопасна (поскольку тип `Bot` пуст, вычисление `t1` все равно не может привести к обычному результату), но по такому правилу получат типы некоторые термы, не являющиеся типизируемыми согласно декларативным правилам; оно сломало бы теорему 16.2.4.

17.3.1. РЕШЕНИЕ: Для решения задачи достаточно перевести на ML алгоритмы из упражнения 16.3.2.

```
let rec join tyS tyT =
  match (tyS, tyT) with
    (TyArr(tyS1, tyS2), TyArr(tyT1, tyT2)) →
      (try TyArr(meet tyS1 tyT1, join tyS2 tyT2)
       with Not_found → TyTop)
    | (TyBool, TyBool) →
        TyBool
    | (TyRecord(fs), TyRecord(ft)) →
        let labelsS = List.map (fun (li, _) → li) fs in
        let labelsT = List.map (fun (li, _) → li) ft in
        let commonLabels =
          List.find_all (fun l → List.mem l labelsT) labelsS in
        let commonFields =
          List.map (fun li →
            let tySi = List.assoc li fs in
            let tyTi = List.assoc li ft in
            (li, join tySi tyTi))
        commonLabels in
        TyRecord(commonFields)
    | _ →
```

```
TyTop

and meet tyS tyT =
  match (tyS,tyT) with
    (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) →
      TyArr(join tyS1 tyT1, meet tyS2 tyT2)
  | (TyBool,TyBool) →
    TyBool
  | (TyRecord(fS), TyRecord(fT)) →
    let labelsS = List.map (fun (li,_) → li) fS in
    let labelsT = List.map (fun (li,_) → li) fT in
    let allLabels =
      List.append
        labelsS
        (List.find_all
          (fun l → not (List.mem l labelsS)) labelsT) in
    let allFields =
      List.map (fun li →
        if List.mem li allLabels then
          let tySi = List.assoc li fS in
          let tyTi = List.assoc li fT in
          (li, meet tySi tyTi)
        else if List.mem li labelsS then
          (li, List.assoc li fS)
        else
          (li, List.assoc li fT))
        allLabels in
      TyRecord(allFields)

  | _ →
    raise Not_found

let rec typeof t =
  match t with
  ...
  | TmTrue(fi) →
    TyBool
  | TmFalse(fi) →
    TyBool
  | TmIf(fi,t1,t2,t3) →
    if subtype (typeof ctx t1) TyBool then
      join (typeof ctx t2) (typeof ctx t3)
    else error fi "условие в условном выражении не типа Bool"
```

17.3.2. РЕШЕНИЕ: См. интерпретатор rcdsubbot.

18.6.1. РЕШЕНИЕ:

```
DecCounter = {get:Unit → Nat, inc:Unit → Unit, reset:Unit →
  Unit,
  dec:Unit → Unit};

decCounterClass =
```

```

λr:CounterRep.
let super = resetCounterClass r in
{get   = super.get,
 inc   = super.inc,
 reset = super.reset,
 dec   = λ_:Unit. r.x:=pred(!(r.x))};

```

18.7.1. РЕШЕНИЕ:

```

BackupCounter2 = {get:Unit → Nat, inc:Unit → Nat,
                  reset:Unit → Unit, backup: Unit → Unit,
                  reset2:Unit → Unit, backup2: Unit → Unit};
BackupCounterRep2 = {x: Ref Nat, b: Ref Nat, b2: Ref Nat};

backupCounterClass2 =
λr:BackupCounterRep2.
let super = backupCounterClass r in
{get = super.get, inc = super.inc,
 reset = super.reset, backup = super.backup,
 reset2 = λ_:Unit. r.x:=!(r.b2),
 backup2 = λ_:Unit. r.b2:=!(r.x)};

```

18.11.1. РЕШЕНИЕ:

```

instrCounterClass =
λr:InstrCounterRep.
λself:Unit → InstrCounter.
λ_:Unit.
let super = setCounterClass r self unit in
{get = λ_:Unit. (r.a:=succ(!!(r.a)); super.get unit),
 set = λi:Nat. (r.a:=succ(!!(r.a)); super.set i),
 inc = super.inc,
 accesses = λ_:Unit. !(r.a)};

ResetInstrCounter = {get:Unit → Nat, set:Nat → Unit,
                     inc:Unit → Unit, accesses:Unit → Nat,
                     reset:Unit → Unit};

resetInstrCounterClass =
λr:InstrCounterRep.
λself:Unit → ResetInstrCounter.
λ_:Unit.
let super = instrCounterClass r self unit in
{get = super.get,
 set = super.set,
 inc = super.inc,
 accesses = super.accesses,
 reset = λ_:Unit. r.x:=0};

BackupInstrCounter = {get:Unit → Nat, set:Nat → Unit,
                      inc:Unit → Unit, accesses:Unit → Unit,
                      backup:Unit → Unit, reset:Unit → Unit};

```

```
BackupInstrCounterRep = {x: Ref Nat, a: Ref Nat, b: Ref Nat};

backupInstrCounterClass =
  λr:BackupInstrCounterRep.
    λself:Unit → BackupInstrCounter.
      λ_:Unit.
        let super = resetInstrCounterClass r self unit in
          {get = super.get,
           set = super.set,
           inc = super.inc,
           accesses = super.accesses,
           reset = λ_:Unit. r.x:=!(r.b),
           backup = λ_:Unit. r.b:=!(r.x)}; 

newBackupInstrCounter =
  λ_:Unit. let r = {x=ref 1, a=ref 0, b=ref 0} in
            fix (backupInstrCounterClass r) unit;
```

18.13.1. РЕШЕНИЕ: Чтобы провести проверку на тождественность, можно, например, использовать ссылочные ячейки. Расширим внутреннее представление наших объектов переменной экземпляра `id` типа `Ref Nat`

```
IdCounterRep = {x:Ref Nat, id: Ref (Ref Nat)};
```

и добавим метод `id`, который просто возвращает поле `id`:

```
IdCounter = {get:Unit → Nat, inc:Unit → Unit, id:Unit →
(Ref Nat)};
idCounterClass =
  λr:IdCounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x:=succ(!(r.x)),
     id = λ_:Unit. !(r.id)};
```

Теперь функция `sameObject` может просто взять два объекта с методами `id` и проверить, одинаковые ли ссылки возвращают эти два метода `id`.

```
sameObject =
  λa:{id:Unit →(Ref Nat)}. λb:{id:Unit →(Ref Nat)}.
  ((b.id unit) := 1,
   (a.id unit) := 0,
   iszero (!(b.id unit)));
```

Фокус состоит в том, что можно легко проверить, указывают ли две ссылки на одно и то же: мы делаем содержимое второй ссылки ненулевым, потом присваиваем ноль первой ссылке, а затем смотрим содержимое второй и проверяем, не стало ли оно нулем.

19.4.1. РЕШЕНИЕ: Поскольку каждое объявление класса должно содержать выражение `extends`, а циклическая зависимость между этими выражениями запрещена, то цепочка выражений `extends`, ведущая от каждого класса, должна в конце концов привести к `Object`.

19.4.2. РЕШЕНИЕ: Один из очевидных способов улучшения языка заключается в объединении трех правил для приведений в одно:

$$\frac{\Gamma \vdash t_0 : D}{\Gamma \vdash (C)t_0 : C} \quad (\text{T-CAST})$$

и отказе от «глупого приведения». Другой способ улучшения сводится к удалению конструкторов, поскольку они все равно ничего не делают.

19.4.6. РЕШЕНИЕ:

1. Формулировка правил для интерфейсов в FJ труда не представляет.
2. Допустим, мы определяем следующие интерфейсы:

```
interface A {}
interface B {}
interface C extends A,B {}
interface D extends A,B {}
```

В такой ситуации C и D имеют в качестве общих верхних граней как A, так и B, но не имеют точной верхней грани.

3. Вместо стандартного алгоритмического правила для условных выражений

$$\frac{\Gamma \vdash t_1 : \text{boolean} \quad \Gamma \vdash t_2 : E_2 \quad \Gamma \vdash t_3 : E_3}{\Gamma \vdash t_1 ? t_2 : t_3 : E_2 \vee E_3}$$

в Java используются следующие ограниченные правила:

$$\frac{\begin{array}{cccc}\Gamma \vdash t_1 : \text{boolean} & \Gamma \vdash t_2 : E_2 & \Gamma \vdash t_3 : E_3 & \Gamma \vdash E_2 <: E_3\end{array}}{\Gamma \vdash t_1 ? t_2 : t_3 : E_3}$$

$$\frac{\begin{array}{cccc}\Gamma \vdash t_1 : \text{boolean} & \Gamma \vdash t_2 : E_2 & \Gamma \vdash t_3 : E_3 & \Gamma \vdash E_3 <: E_2\end{array}}{\Gamma \vdash t_1 ? t_2 : t_3 : E_2}$$

Интуитивно эти правила корректны, но они плохо сочетаются с операционной семантикой с малым шагом, принятой в FJ — свойство сохранения типов оказывается ложным! (Нетрудно построить пример, демонстрирующий это.)

19.4.7. РЕШЕНИЕ: Как ни странно, поддержка `super` оказывается сложнее, чем поддержка `self`, поскольку требуется как-то запоминать, из какого класса взят «выполняемый в данный момент» метод. Есть по крайней мере два способа это сделать:

1. Снабжать термы аннотацией, указывающей, где нужно искать значение `super`.
2. Добавить шаг предварительной обработки, на котором переписывается вся таблица классов, и все упоминания `super` переводятся в упоминания `this` с «дополненными» именами, которые указывают, из какого класса они взяты.

19.5.1. РЕШЕНИЕ: Прежде чем представить полное доказательство, введем несколько вспомогательных лемм. Как всегда, самая важная из них, А.14, связывает типизацию и подстановку.

Лемма А.13 Если $mtype(m, D) = \bar{C} \rightarrow C_0$, то $mtype(m, C) = \bar{C} \rightarrow C_0$ для всех $C <: D$.

Доказательство: Прямолинейная индукция по дереву вывода $C <: D$. Заметим, что, независимо от того, определен или не определен метод m в $CT(C)$, $mtype(m, C)$ должен оставаться таким же, как $mtype(m, E)$, где $CT(C) = \text{class } C \text{ extends } E \{ \dots \}$.

Лемма А.14 [Подстановка термов сохраняет типизацию]:

Если $\Gamma, \bar{x}:\bar{B} \vdash t : D$ и $\Gamma \vdash \bar{s} : \bar{A}$, где $\bar{A} <: \bar{B}$, то $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t : C$ для некоторого $C <: D$.

Доказательство: Индукция по дереву вывода $\Gamma, \bar{x}:\bar{B} \vdash t : D$. Интуитивные шаги те же самые, что и в лямбда-исчислении с подтипами; детали, разумеется, немного отличаются. Наиболее интересны последние два варианта.

Вариант T-VAR: $t = x \quad x:D \in \Gamma$

Если $x \notin \bar{x}$, то результат тривиален, поскольку $[\bar{x} \mapsto \bar{s}]x = x$. С другой стороны, если $x = x_i$, а $D = B_i$, то, поскольку $[\bar{x} \mapsto \bar{s}]x = s_i$, достаточно положить $C = A_i$, и доказательство окончено.

Вариант T-FIELD: $t = t_0.f_i \quad \Gamma, \bar{x}:\bar{B} \vdash t_0 : D_0 \quad \text{fields}(D_0) = \bar{C} \bar{f} \quad D = C_i$

По предположению индукции, существует некоторый C_0 , такой что $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 : C_0$ и $C_0 <: D_0$. Нетрудно проверить, что $\text{fields}(C_0) = (\text{fields}(D_0), \bar{D} \bar{g})$ для некоторых $\bar{D} \bar{g}$. Следовательно, по правилу T-FIELD, $\Gamma \vdash ([\bar{x} \mapsto \bar{s}]t_0).f_i : C_i$.

Вариант T-INVK: $t = t_0.m(\bar{t}) \quad \Gamma, \bar{x}:\bar{B} \vdash t_0 : D_0 \quad mtype(m, D_0) = \bar{E} \rightarrow D$
 $\Gamma, \bar{x}:\bar{B} \vdash \bar{t} : \bar{D} \quad \bar{D} <: \bar{E}$

По предположению индукции, существуют такие C_0 и \bar{C} , что:

$$\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 : C_0 \quad C_0 <: D_0 \quad \Gamma \vdash [\bar{x} \mapsto \bar{s}]\bar{t} : \bar{C} \quad \bar{C} <: \bar{D}$$

По лемме А.13, $mtype(m, C_0) = \bar{E} \rightarrow D$. Кроме того, $\bar{C} <: \bar{E}$, по свойству транзитивности $<:$. Следовательно, по правилу T-INVK, $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0.m([\bar{x} \mapsto \bar{s}]\bar{t}) : D$.

Вариант T-NEW: $t = \text{new } D(\bar{t}) \quad \text{fields}(D) = \bar{D} \bar{f} \quad \Gamma, \bar{x}:\bar{B} \vdash \bar{t} : \bar{C} \quad \bar{C} <: \bar{D}$

По предположению индукции, $\Gamma \vdash [\bar{x} \mapsto \bar{s}]\bar{t} : \bar{E}$ для некоторых \bar{E} , причем $\bar{E} <: \bar{C}$. Имеем $\bar{E} <: \bar{D}$ по свойству транзитивности $<:$. Следовательно, по правилу T-NEW, $\Gamma \vdash \text{new } D([\bar{x} \mapsto \bar{s}]\bar{t}) : D$.

Вариант T-UCAST: $t = (D)t_0 \quad \Gamma, \bar{x}:\bar{B} \vdash t_0 : C \quad C <: D$

По предположению индукции, существует такой E , что $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 : E$ и $E <: C$. Имеем $E <: D$ по свойству транзитивности $<:$, что дает нам $\Gamma \vdash (D)([\bar{x} \mapsto \bar{s}]t_0) : D$ по правилу T-UCAST.

Вариант T-DCAST: $t = (D)t_0 \quad \Gamma, \bar{x}:\bar{B} \vdash t_0 : C \quad D <: C \quad D \neq C$

По предположению индукции, существует такой E , что $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 : E$

и $E <: C$. Если $E <: D$ или $D <: E$, то $\Gamma \vdash (D)([\bar{x} \mapsto \bar{s}]t_0) : D$, соответственно, по T-UCAST или T-DCAST. С другой стороны, если $D <: E$ и $E <: D$, то $\Gamma \vdash (D)([\bar{x} \mapsto \bar{s}]t_0) : D$ (с предупреждением о глупости) по правилу T-SCAST.

Вариант T-SCAST: $t = (D)t_0 \quad \Gamma, \bar{x}:\bar{B} \vdash t_0 : C \quad D <: C \quad C <: D$
 По предположению индукции, существует такой E , что $\Gamma \vdash [\bar{x} \mapsto \bar{s}]t_0 : E$ и $E <: C$. Это означает, что $E <: D$. (Чтобы убедиться в этом, заметим, что у каждого класса в FJ есть только один надкласс. Отсюда следует, что, если $E <: C$ и $E <: D$, то либо $C <: D$, либо $D <: C$.) Таким образом, $\Gamma \vdash (D)([\bar{x} \mapsto \bar{s}]t_0) : D$ (с предупреждением о глупости) по правилу T-SCAST.

Лемма A.15 [ОСЛАБЛЕНИЕ]: Если $\Gamma \vdash t : C$, то $\Gamma, x:D \vdash t : C$.

Доказательство: Прямолинейная индукция.

Лемма A.16 Если $mtype(m, C_0) = \bar{D} \rightarrow D$, а $mbody(m, C_0) = (\bar{x}, t)$, то для некоторого D_0 и некоторого $C <: D$ выполняется $C_0 <: D_0$ и $x:D, this:D_0 \vdash t : C$.
Доказательство: Индукция по дереву вывода $mbody(m, C_0)$. Базовый случай (когда m определен внутри C_0) доказывается просто, поскольку m определен в $CT(C_0)$, и из корректности таблицы классов прямо следует, что мы уже вывели $x:\bar{D}, this:C_0 \vdash t : C$ по правилу T-МЕТОД. Шаг индукции также не представляет труда.

Теперь мы можем доказать теорему о типовой безопасности.

Доказательство теоремы 19.5.1: Индукция по дереву вывода $t \rightarrow t'$, с разбором вариантов последнего правила. Обратите внимание на то, как в предпоследнем варианте, T-DCAST, порождаются предупреждения о глупости.

Вариант E-PROJNEW: $t = \text{new } C_0(\bar{v}) . f_i \quad t' = v_i \quad fields(C_0) = \bar{D} \bar{f}$

По форме t мы можем определить, что последним правилом в выводе $\Gamma \vdash t : C$ было T-FIELD, с предпосылкой $\Gamma \vdash \text{new } C_0(\bar{v}) : D_0$, для некоторого D_0 , и что $C = D_i$. Аналогично, последним правилом в выводе $\Gamma \vdash \text{new } C_0(\bar{v}) : D_0$ должно быть T-NEW, с предпосылками $\Gamma \vdash \bar{v} : \bar{C}$ при $\bar{C} <: \bar{D}$ и $D_0 = C_0$. В частности, $\Gamma \vdash v_i : C_i$, что завершает рассмотрение данного варианта, поскольку $C_i <: D_i$.

Вариант E-INVKNEW: $t = (\text{new } C_0(\bar{v})) . m(\bar{u}) \quad t' = [\bar{u}/\bar{x}, \text{new } C_0(\bar{v}) / this]t_0$
 $mbody(m, C_0) = (\bar{x}, t_0)$

Финальными правилами при выводе $\Gamma \vdash t : C$ должны быть T-INVK и T-NEW, с предпосылками

$$\Gamma \vdash \text{new } C_0(\bar{v}) : C_0 \quad \Gamma \vdash \bar{u} : \bar{C} \quad \bar{C} <: \bar{D} \quad mtype(m, C_0) = \bar{D} \rightarrow C$$

По лемме A.16 имеем $\bar{x}:\bar{D}, this:D_0 \vdash t_0 : B$ для некоторых D_0 и B , причем $C_0 <: D_0$ и $B <: C$. По лемме A.15, $\Gamma, \bar{x}:\bar{D}, this:D_0 \vdash t_0 : B$. Отсюда, по лемме A.14, $\Gamma \vdash [\bar{x} \mapsto \bar{u}, this \mapsto \text{new } C_0(\bar{v})]t_0 : E$ для некоторого $E <: B$. По свойству транзитивности $<:$ получаем $E <: C$. Теперь достаточно принять $C' = E$, чтобы завершить доказательство.

Вариант E-CASTNEW: $t = (D)(\text{new } C_0(\bar{v})) \quad C_0 <: D \quad t' = \text{new } C_0(\bar{v})$

Вывод $\Gamma \vdash (D)(\text{new } C_0(\bar{v})) : C$ должен завершаться правилом T-UCAST, поскольку, если бы он кончался правилами T-SCAST или T-DCAST, получалось бы противоречие с предположением $C_0 <: D$. Предпосылки T-UCAST дают нам $\Gamma \vdash \text{new } C_0(\bar{v}) : C_0$ и $D = C$, что завершает рассмотрение этого варианта.

Варианты с правилами соответствия не представляют труда. Мы рассмотрим только одно:

Вариант RC-CAST: $t = (D)t_0 \quad t' = (D)t'_0 \quad t_0 \rightarrow t'_0$

Существует три подварианта, в зависимости от последнего использованного правила типизации.

Подвариант T-UCAST: $\Gamma \vdash t_0 : C_0 \quad C_0 <: D \quad D = C$

По предположению индукции, $\Gamma \vdash t'_0 : C'_0$ для некоторого $C'_0 <: C_0$. По свойству транзитивности $<:$, $C'_0 <: C$. Следовательно, по правилу T-UCAST, $\Gamma \vdash (C)t'_0 : C$ (без дополнительного *предупреждения о глупости*).

Подвариант T-DCAST: $\Gamma \vdash t_0 : C_0 \quad D <: C_0 \quad D = C$

По предположению индукции, $\Gamma \vdash t'_0 : C'_0$ для некоторого $C'_0 <: C_0$. Если $C'_0 <: C$ либо $C <: C'_0$, имеем $\Gamma \vdash (C)t'_0 : C$ по правилу T-UCAST либо T-DCAST (без дополнительного *предупреждения о глупости*). Если же $C'_0 \triangleleft C$ и $C \triangleleft C'_0$, то $\Gamma \vdash (C)t'_0 : C$ с *предупреждением о глупости* по правилу T-SCAST.

Подвариант T-SCAST: $\Gamma \vdash t_0 : C_0 \quad D \triangleleft C_0 \quad C_0 \triangleleft D \quad D = C$

По предположению индукции, $\Gamma \vdash t'_0 : C'_0$ для некоторого $C'_0 <: C_0$. Значит, выполняются также $C'_0 \triangleleft C$ и $C \triangleleft C'_0$. Следовательно, $\Gamma \vdash (C)t'_0 : C$ с *предупреждением о глупости*.

20.1.1. РЕШЕНИЕ:

```

Tree = μX. <leaf:Unit, node:{Nat,X,X}>;
leaf = <leaf=unit> as Tree;

▷ leaf : Tree

node = λn:Nat. λt1:Tree. λt2:Tree. <node={n,t1,t2}> as Tree;

▷ node : Nat → Tree → Tree → Tree

isleaf = λl:Tree. case l of <leaf=u> ⇒ true | <node=p> ⇒
false;

▷ isleaf : Tree → Bool

label = λl:Tree. case l of <leaf=u> ⇒ 0 | <node=p> ⇒ p.1;

▷ label : Tree → Nat

```

```

left = λl:Tree. case l of <leaf=u> => leaf | <node=p> => p.2;
▷ left : Tree → Tree

right = λl:Tree. case l of <leaf=u> => leaf | <node=p> =>
p.3;

▷ right : Tree → Tree

append = fix (λf:NatList → NatList → NatList.
    λl1:NatList. λl2:NatList.
    if isnil l1 then l2 else
    cons (hd l1) (f (tl l1) l2));

▷ append : NatList → NatList → NatList

preorder = fix (λf:Tree → NatList. λt:Tree.
    if isleaf t then nil else
    cons (label t)
        (append (f (left t)) (f (right t))));

▷ preorder : Tree → NatList

t1 = node 1 leaf leaf;
t2 = node 2 leaf leaf;
t3 = node 3 t1 t2;
t4 = node 4 t3 t3;
l = preorder t4;
hd l;

▷ 4 : Nat

hd (tl l)

▷ 3 : Nat

hd (tl (tl l))

▷ 1 : Nat

```

20.1.2. РЕШЕНИЕ:

```

fib = fix (λf: Nat → Nat → Stream. λm:Nat. λn:Nat. λ_:Unit.
    {n, f n (plus m n)}) 0 1;

▷ fib : Stream

```

20.1.3. РЕШЕНИЕ:

```

Counter = μC. {get:Nat, inc:Unit → C, dec:Unit → C,
    reset:Unit → C, backup:Unit → C};

c = let create =

```

```

fix (λcr: {x:Nat,b:Nat} → Counter. λs: {x:Nat,b:Nat}.
    get      = s.x,
    inc      = λ_:Unit. cr {x=succ(s.x),b=s.b},
    dec      = λ_:Unit. cr {x=pred(s.x),b=s.b},
    backup   = λ_:Unit. cr {x=s.x,b=s.x},
    reset   = λ_:Unit. cr {x=s.b,b=s.b} )
in create {x=0,b=0};

▷ c : Counter

```

20.1.4. РЕШЕНИЕ:

```

D = μX. <nat:Nat, bool:Boolean, fn:X → X>;

lam = λf:D → D. <fn=f> as D;
ap = λf:D. λa:D.
    case f of
        <nat=n> ⇒ divergeD unit
        | <bool=b> ⇒ divergeD unit
        | <fn=f> ⇒ f a;
ifd = λb:D. λt:D. λe:D.
    case b of
        <nat=n> ⇒ divergeD unit
        | <bool=b> ⇒ (if b then t else e)
        | <fn=f> ⇒ divergeD unit;
tru = <bool=true> as D;
fls = <bool=false> as D;
ifd fls one zro;

▷ <nat=0> as D : D

ifd fls one fls

▷ <bool=false> as D : D

```

Читатели, обеспокоенные тем, что в этой системе мы можем закодировать неверно типизированные термы, должны заметить, что мы всего лишь построили *структуру данных* для представления объектного языка бестиповых термов внутри метаязыка простого типизированного лямбда-исчисления с рекурсивными типами. То, что это оказалось возможным, не более удивительно, чем тот факт (используемый нами во всех главах, описывающих реализации), что термы различных типизированных и бестиповых вариантов лямбда-исчислений могут быть представлены в виде структур данных ML.

20.1.5. РЕШЕНИЕ:

```

lam = λf:D → D. <fn=f> as D;
ap = λf:D. λa:D. case f of
    <nat=n> ⇒ divergeD unit
    | <fn=f> ⇒ f a
    | <rcc=r> ⇒ divergeD unit;
rcc = λfields:Nat → D. <rcc=fields> as D;
prj = λf:D. λn:Nat. case f of

```

```

<nat=n> => diverge_D unit
| <fn=f> => diverge_D unit
| <rcd=r> => r n;
myrcd = rcd (λn:Nat. if iszero n then zro
                  else if iszero (pred n) then one
                  else diverge_D unit);

```

20.2.1. РЕШЕНИЕ: Вот самые интересные примеры в изорекурсивной форме:

```

Hungry = μA. Nat → A;
f = fix (λf: Nat → Hungry. λn:Nat. fold [Hungry] f);
ff = fold [Hungry] f;
ff1 = (unfold [Hungry] ff) 0;
ff2 = (unfold [Hungry] ff1) 2;

fix_T =
  λf:T → T.
    (λx:(μA.A → T). f ((unfold [μA.A → T] x) x))
      (fold [μA.A → T] (λx:(μA.A → T). f ((unfold [μA.A → T] x) x)));

D = μX. X → X;
lam = λf:D → D. fold [D] f;
ap = λf:D. λa:D. (unfold [D] f) a;

Counter = μC. {get:Nat, inc:Unit → C};
c = let create = fix (λcr: {x:Nat} → Counter. λs: {x:Nat}.
  fold [Counter]
    {get = s.x,
     inc = λ_:Unit. cr {x=succ(s.x)})})
  in create {x=0};
c1 = (unfold [Counter] c).inc unit;
(unfold [Counter] c1).get;

```

21.1.7. РЕШЕНИЕ:

$$\begin{array}{llll}
 E_2(\emptyset) & = & \{a\} & E_2(\{a,b\}) = \{a,c\} \\
 E_2(\{a\}) & = & \{a\} & E_2(\{a,c\}) = \{a,b\} \\
 E_2(\{b\}) & = & \{a\} & E_2(\{b,c\}) = \{a,b\} \\
 E_2(\{c\}) & = & \{a,b\} & E_2(\{a,b,c\}) = \{a,b,c\}
 \end{array}$$

E_2 -замкнутыми множествами являются $\{a\}$ и $\{a,b,c\}$. E_2 -консистентны множества \emptyset , $\{a\}$ и $\{a,b,c\}$. Наименьшая неподвижная точка E_2 равна $\{a\}$. Наибольшая неподвижная точка равна $\{a,b,c\}$.

21.1.9. РЕШЕНИЕ: Чтобы доказать принцип обыкновенной индукции на натуральных числах, мы действуем следующим образом. Определяем порождающую функцию $F \in \mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N})$ как

$$F(X) = \{0\} \cup \{i+1 \mid i \in X\}$$

Теперь предположим, что у нас есть предикат (т. е. множество чисел) P , такой, что $P(0)$ — истина, и из истинности $P(i)$ следует истинность $P(i+1)$. Тогда из

определения F нетрудно видеть, что $F(P) \subseteq P$, т. е. что множество P является F -замкнутым. Согласно принципу индукции, $\mu F \subseteq P$. Однако μF равняется всему множеству натуральных чисел (в сущности, это можно воспринимать как *определение* множества натуральных чисел), так что $P(n)$ истинно для всех $n \in \mathbb{N}$.

В случае лексикографической индукции определяем $F \in \mathcal{P}(\mathbb{N} \times \mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N})$ так:

$$F(X) = \{(m, n) \mid \forall (m', n') < (m, n), (m', n') \in X\}$$

Теперь допустим, что у нас есть предикат (т. е. множество пар чисел) P такой, что всегда, когда $P(m', n')$ истинно для всех $(m', n') < (m, n)$, мы имеем также, что $P(m, n)$ тоже истинно. Как и раньше, из определения F нетрудно увидеть, что $F(P) \subseteq P$, т. е., что множество P является F -замкнутым. Согласно принципу индукции, $\mu F \subseteq P$. Чтобы завершить доказательство, нужно проверить, что μF действительно совпадает с множеством всех пар чисел (это единственный нетривиальный шаг рассуждения). Это можно доказать в два шага. Сначала заметим, что множество $\mathbb{N} \times \mathbb{N}$ является F -замкнутым (это непосредственно следует из определения F). Во-вторых, докажем, что ни одно собственное подмножество $\mathbb{N} \times \mathbb{N}$ не является F -замкнутым — т. е., что $\mathbb{N} \times \mathbb{N}$ является наименьшим F -замкнутым множеством. Чтобы увидеть это, предположим, что имеется некоторое меньшее F -замкнутое множество Y , и пусть (m, n) будет наименьшей парой, *не* принадлежащей Y ; согласно определению F мы видим, что $F(Y) \not\subseteq Y$, т. е. множество Y незамкнуто — мы пришли к противоречию.

21.2.2. РЕШЕНИЕ: Определим *дерево* как частичную функцию $T \in \{1, 2\}^* \rightarrow \{\rightarrow, \times, \text{Top}\}$, которая удовлетворяет следующим ограничениям:

- $T(\bullet)$ определена;
- если определена $T(\pi, \sigma)$, то определена и $T(\pi)$.

Заметим, что символы $\rightarrow, \times, \text{Top}$ могут без ограничений упоминаться в вершинах дерева — например, вершина с меткой *Top* может иметь нетривиальных потомков, и т. п. Как и в §21.2, мы используем символы \rightarrow, \times и *Top* также и в качестве имен операторов над деревьями.

Множество всех деревьев берется как универсум \mathcal{U} . Порождающая функция F основана на уже знакомой нам грамматике для типов:

$$\begin{aligned} F(X) &= \{\text{Top}\} \\ &\cup \{T_1 \times T_2 \mid T_1, T_2 \in X\} \\ &\cup \{T_1 \rightarrow T_2 \mid T_1, T_2 \in X\} \end{aligned}$$

Из определений \mathcal{T} и \mathcal{U} нетрудно видеть, что $\mathcal{T} \subseteq \mathcal{U}$, так что имеет смысл сравнение множеств в интересующих нас уравнениях, $\mathcal{T} = \nu F$ и $\mathcal{T}_f = \mu F$. Остается показать, что эти уравнения истинны.

$\mathcal{T} \subseteq \nu F$ следует по принципу коиндукции из того, что множество \mathcal{T} является F -консистентным. Чтобы получить $\nu F \subseteq \mathcal{T}$, требуется для всякого $T \in \nu F$ проверить два последних условия из определения 21.2.1. Это можно сделать при помощи индукции по длине π .

$\mu F \subseteq \mathcal{T}_f$ следует по принципу индукции из того, что множество \mathcal{T}_f является F -замкнутым. Чтобы получить $\mathcal{T}_f \subseteq \mu F$, докажем индукцией по размеру T , что из $T \in \mathcal{T}_f$ следует $T \in \mu F$. (Размер дерева можно определить как длину наибольшей последовательности $\pi \in \{1, 2\}^*$, для которой определена $T(\pi)$.)

21.3.3. Решение: Пара $(\text{Top}, \text{Top} \times \text{Top})$ не лежит в νS . Чтобы увидеть это, заметим, что, по определению S , эта пара не принадлежит $S(X)$ ни для какого X . Таким образом, нет ни одного S -консистентного множества, содержащего эту пару; в частности, таковым не является множество νS (которое S -консистентно).

21.3.4. Решение: В качестве примера пары типов, связанных отношением νS , но не μS , можно взять пару (T, T) для любого бесконечного типа T . Рассмотрим множество пар $R = \{(Q, Q) \mid \text{для } Q, \text{ являющегося поддеревом } T\}$. Рассмотрев определение S , легко получаем $R \subseteq S(R)$, а применение принципа коиндукции дает нам $R \subseteq \nu S$. Отсюда $(T, T) \in \nu S$, поскольку $(T, T) \in R$. С другой стороны, $R \notin \mu S$, поскольку μS связывает исключительно конечные типы* — это можно установить, обозначив множество всех пар конечных типов через R' и получив, что $\mu S \subseteq R'$ по принципу индукции.

Не существует пар конечных типов (S, T) , связанных отношением νS_f , но не μS_f , поскольку эти две неподвижные точки совпадают. Это следует из того, что для любых $S, T \in \mathcal{T}_f$ из $(S, T) \in \mu S_f$ следует $(S, T) \in \nu S_f$. (Поскольку T представляет собой конечное дерево, последнее утверждение можно получить индукцией по сумме размеров S и T . Нужно рассмотреть варианты, когда T равен Top , $T_1 \times T_2$, $T_1 \rightarrow T_2$, использовать определение S_f и равенства $S_f(\nu S_f) = \nu S_f$ и $S_f(\mu S_f) = \mu S_f$.)

21.3.8. Решение: Сначала определим отношение тождества для древовидных типов: $I = \{(T, T) \mid T \in \mathcal{T}\}$. Если мы покажем, что I является S -консистентно, то принцип коиндукции позволит нам заключить, что $I \subseteq \nu S$ — то есть, что νS рефлексивно. Чтобы показать S -консистентность I , возьмем элемент $(T, T) \in I$, и рассмотрим варианты его формы. Сначала допустим, что $T = \text{Top}$. Тогда $(T, T) = (\text{Top}, \text{Top})$, а эта пара по определению принадлежит $S(I)$. Теперь допустим, что $T = T_1 \times T_2$. Тогда, поскольку $(T_1, T_1), (T_2, T_2) \in I$, с помощью определения S имеем $(T_1 \times T_2, T_1 \times T_2) \in S(I)$. Аналогично для $T = T_1 \rightarrow T_2$.

21.4.2. Решение: Согласно принципу коиндукции достаточно показать, что множество $\mathcal{U} \times \mathcal{U}$ является F^{TR} -консистентным, т. е. $\mathcal{U} \times \mathcal{U} \subseteq F^{TR}(\mathcal{U} \times \mathcal{U})$. Допустим, $(x, y) \in \mathcal{U} \times \mathcal{U}$. Возьмем любое $z \in \mathcal{U}$. Тогда $(x, z), (z, y) \in \mathcal{U} \times \mathcal{U}$, таким образом, по определению F^{TR} , имеем $(x, y) \in F^{TR}(\mathcal{U} \times \mathcal{U})$.

21.5.2. Решение: Чтобы проверить инвертируемость, просто рассмотрим определения S_f и S и убедимся в том, что каждое множество $G_{(s,t)}$ содержит не более одного элемента.

В определениях S_f и S каждый вариант явным образом указывает форму поддерживаемого элемента и состав его множества поддержки, так что нетрудно выписать support_{S_f} и support_S . (Сравните с функцией поддержки для S_m из определения 21.8.4.)

*В исправлениях к тексту книги автор опровергает это утверждение: «Это не совсем верно: все типы, в том числе и бесконечные, являются подтипом типа Top . Здесь необходимо более сложное утверждение.» — прим. перев.

21.5.4. РЕШЕНИЕ:

$$\frac{i}{h} \quad \frac{a}{a} \quad \frac{b}{d} \quad \frac{c}{e} \quad \frac{d}{b} \quad \frac{e}{\bar{b}} \quad \frac{f}{c} \quad \frac{g}{f} \quad \frac{g}{\bar{g}}$$

21.5.6. РЕШЕНИЕ: Нет, путь от $x \in \nu F \setminus \mu F$ не обязательно ведет к циклу в графе поддержки; он может вести и к бесконечной цепочке. Рассмотрим, например, $F \in \mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N})$, определяемую уравнением $F(X) = \{0\} \cup \{n \mid n + 1 \in X\}$. Здесь $\mu F = \{0\}$, а $\nu F = \mathbb{N}$. Кроме того, для всякого $n \in \nu F \setminus \mu F$, то есть для всякого $n > 0$ $\text{support}(n) = \{n + 1\}$, и таким образом порождается бесконечная цепочка.

21.5.13. РЕШЕНИЕ: Рассмотрим сначала частичную корректность. Доказательство в каждой части ведется индукцией по рекурсивной структуре вызовов при прогоне алгоритма.

1. Из определения lfp нетрудно видеть, что эта функция возвращает значение *истина* в двух случаях. Если $\text{lfp}(X) = \text{истина}$, потому что $X = \emptyset$, то утверждение $X \subseteq \mu F$ тривиально верно. С другой стороны, если $\text{lfp}(X) = \text{истина}$, потому что $\text{lfp}(\text{support}(X)) = \text{истина}$, то, по предположению индукции, $\text{support}(X) \subseteq \mu F$, а отсюда лемма 21.5.8 дает нам $X \subseteq \mu F$.
2. Если $\text{lfp}(X) = \text{ложь}$, потому что $\text{support}(X) \uparrow$, то $X \not\subseteq \mu F$ по лемме 21.5.8. В противном случае $\text{lfp}(X) = \text{ложь}$, потому что $\text{lfp}(\text{support}(X)) = \text{ложь}$, и по предположению индукции, $\text{support}(X) \not\subseteq \mu F$. По лемме 21.5.8, $X \not\subseteq \mu F$.

Затем нам нужно охарактеризовать порождающие функции F , для которых lfp гарантированно завершается для всех конечных входных данных. Здесь будет полезно терминологическое нововведение. Если имеется порождающая функция $F \in \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ с конечным числом состояний, то частичная функция $\text{height}_F \in \mathcal{U} \rightarrow \mathbb{N}$ (или просто height , «высота») есть наименьшая частичная функция, удовлетворяющая следующему условию:²

$$\text{height}(x) = \begin{cases} 0 & \text{если } \text{support}(x) = \emptyset \\ 0 & \text{если } \text{support}(x) \uparrow \\ 1 + \max\{\text{height}(y) \mid y \in \text{support}(x)\} & \text{если } \text{support}(x) \neq \emptyset \end{cases}$$

(Заметим, что $\text{height}(x)$ не определена, если x либо принадлежит циклу доступности, либо зависит от элемента такого цикла.) Будем говорить, что порождающая функция F имеет *конечную высоту*, если функция height_F всюду определена. Нетрудно проверить, что если $y \in \text{support}(x)$, и определены как $\text{height}(x)$, так и $\text{height}(y)$, то $\text{height}(y) < \text{height}(x)$.

Теперь мы можем сказать, что если функция F имеет конечное число состояний и конечную высоту, то $\text{lfp}(X)$ завершается для всякого конечного

²Заметим, что такой способ задания функции height легко переформулировать как наименьшую неподвижную точку монотонной функции на отношениях, представляющих частичные функции.

входного множества $X \subseteq \mathcal{U}$. Чтобы убедиться в этом, заметим, что поскольку у F конечное число состояний, то в каждом рекурсивном вызове $\text{lfp}(Y)$, происходящем от исходного вызова $\text{lfp}(X)$, множество Y конечно. Поскольку F имеет конечную высоту, функция $h(Y) = \max\{\text{height}(y) \mid y \in Y\}$ правильно определена. Поскольку $h(Y)$ уменьшается при каждом рекурсивном вызове и всегда неотрицательна, она может служить мерой завершения для lfp .

21.8.5. Решение: Определение S_d совпадает с определением S_m , за исключением того, что последний вариант не содержит условий $T \neq \mu X.T_1$ и $T \neq \text{Top}$. Чтобы убедиться в неинвертируемости S_d , заметим, что множество $G_{(\mu X.\text{Top}, \mu Y.\text{Top})}$ содержит два порождающих множества: $\{\text{(Top, } \mu Y.\text{Top)}\}$ и $\{(\mu X.\text{Top}, \text{Top})\}$ (сравните с тем, как то же множество выглядит для функции S_m).

Поскольку все варианты, кроме последнего, в определениях S_d и S_m совпадают, а последний вариант S_m представляет собой ограничение последнего варианта S_d , включение $\nu S_m \subseteq \nu S_d$ очевидно. В другом направлении включение $\nu S_d \subseteq \nu S_m$ можно доказать с помощью принципа коиндукции и следующей леммы, устанавливающей, что множество νS_d является S_m -консistentным.

Лемма A.17 Для всякой пары μ -типов S, T , если $(S, T) \in \nu S_d$, то $(S, T) \in S_m(\nu S_d)$.

Набросок доказательства: Индукция по $k = \mu\text{-height}(S)$. Такая индукция выражает неформальную идею о том, что любой вывод $(S, T) \in \nu S_d$ можно перевести в более короткий вывод того же самого утверждения, и этот последний всегда будет одновременно выводом утверждения $(S, T) \in \nu S_m$. Из ограничений на правило левой μ -свертки следует, что полученный в результате трансформации вывод будет обладать тем свойством, что всякая последовательность применений правил μ -свертки начинается с последовательности левых μ -сверток, за которыми следует последовательность правых μ -сверток.

21.9.2. Решение:

$$\begin{array}{c} \frac{}{T \sqsubseteq T} \quad \frac{S \sqsubseteq T_1}{S \sqsubseteq T_1 \times T_2} \quad \frac{S \sqsubseteq T_2}{S \sqsubseteq T_1 \times T_2} \\ \frac{S \sqsubseteq T_1}{S \sqsubseteq T_1 \rightarrow T_2} \quad \frac{S \sqsubseteq T_2}{S \sqsubseteq T_1 \rightarrow T_2} \quad \frac{S \sqsubseteq [X \mapsto \mu X.T]T}{S \sqsubseteq \mu X.T} \end{array}$$

(Отметим тот интересный факт, что порождающая функция TD отличается от других функций, рассмотренных нами в этой главе: она не обратима. Например, $B \sqsubseteq A \times B \rightarrow B \times C$ поддерживается двумя множествами, $\{B \sqsubseteq A \times B\}$ и $\{B \sqsubseteq B \times C\}$, ни одно из которых не является подмножеством другого.)

21.9.7. Решение: Все правила для BU совпадают с правилами для TD из решения к упражнению 21.9.2 за исключением правила для типов, начинающихся с конструкции μ :

$$\frac{S \leq T}{[X \mapsto \mu X.T]S \leq \mu X.T}$$

21.11.1. РЕШЕНИЕ: Таких пар типов много. Тривиальным примером может служить $\mu X.T$ и $[X \mapsto \mu X.T]T$ почти для любого T . Более интересный пример — $\mu X.\text{Nat} \times (\text{Nat} \times X)$ и $\mu X.\text{Nat} \times X$.

22.3.9. РЕШЕНИЕ: Вот основные правила алгоритмического порождения ограничений:

$$\frac{\Gamma(x) = T}{\Gamma \vdash_F x : T|_F \{ \}} \quad (\text{CT-VAR})$$

$$\frac{\Gamma, x:T_1 \vdash_F t_2 : T_2|_{F'} \mathcal{C} \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash_F \lambda x:T_1. t_2 : T_1 \rightarrow T_2|_{F'} \mathcal{C}} \quad (\text{CT-Abs})$$

$$\frac{\Gamma \vdash_F t_1 : T_1|_{F'} \mathcal{C}_1 \quad \Gamma \vdash_{F'} t_2 : T_2|_{F''} \mathcal{C}_2 \quad F'' = X, F'''}{\Gamma \vdash_F t_1 \cdot t_2 : X|_{F'''} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 = T_2 \rightarrow X\}} \quad (\text{CT-APP})$$

Остальные правила устроены так же. Эквивалентность исходных правил и их алгоритмического представления формулируется следующим образом:

1. [КОРРЕКТНОСТЬ]: Если $\Gamma \vdash_F t : T|_{F'} \mathcal{C}$, а переменные, упомянутые в Γ и t , не встречаются в F , то $\Gamma \vdash t : T|_{F \setminus F'} \mathcal{C}$.
2. [ПОЛНОТА]: Если $\Gamma \vdash t : T|_{\mathcal{X}} \mathcal{C}$, то существует такая перестановка F имен переменных из \mathcal{X} , что $\Gamma \vdash_F t : T|_{\emptyset} \mathcal{C}$

Оба утверждения доказываются прямолинейной индукцией по деревьям вывода. Для случая терма-применения в части 1 оказывается полезна следующая лемма:

Если типовые переменные, встречающиеся в Γ и t , не содержатся в F' , а $\Gamma \vdash_F t : T|_{F'} \mathcal{C}$, то типовые переменные, встречающиеся в T и \mathcal{C} , не содержатся в $F \setminus F'$.

Для соответствующего случая в части 2 используется следующая лемма:

Если $\Gamma \vdash_F t : T|_{F'} \mathcal{C}$, то $\Gamma \vdash_{F,G} t : T|_{F',G} \mathcal{C}$, где G — произвольная последовательность еще не использованных имен переменных.

22.3.10. РЕШЕНИЕ: Если представлять множества ограничений как списки пар типов, то алгоритм порождения ограничений будет простым переводом на ML правил вывода из решения к упражнению 22.3.9.

```
let rec recon ctx nextuvar t = match t with
  | TmVar(fi,i,_) →
    let tyT = getTypeFromContext fi ctx i in
    (tyT, nextuvar, [])
  | TmAbs(fi, x, tyT1, t2) →
    let ctx' = addbinding ctx x (VarBind(tyT1)) in
    let (tyT2, nextuvar2, constr2) = recon ctx' nextuvar t2 in
    (TyArr(tyT1, tyT2), nextuvar2, constr2)
  | TmApp(fi,t1,t2) →
```

```

let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
let (tyT2,nextuvar2,constr2) = recon ctx nextuvar1 t2 in
let NextUVar(tyX,nextuvar') = nextuvar2() in
let newconstr = [(tyT1,TyArr(tyT2,TyId(tyX)))] in
((TyId(tyX)), nextuvar',
List.concat [newconstr; constr1; constr2])
| TmZero(fi) → (TyNat, nextuvar, [])
| TmSucc(fi,t1) →
let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
(TyNat, nextuvar1, (tyT1,TyNat)::constr1)
| TmPred(fi,t1) →
let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
(TyNat, nextuvar1, (tyT1,TyNat)::constr1)
| TmIsZero(fi,t1) →
let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
(TyBool, nextuvar1, (tyT1,TyNat)::constr1)
| TmTrue(fi) → (TyBool, nextuvar, [])
| TmFalse(fi) → (TyBool, nextuvar, [])
| TmIf(fi,t1,t2,t3) →
let (tyT1,nextuvar1,constr1) = recon ctx nextuvar t1 in
let (tyT2,nextuvar2,constr2) = recon ctx nextuvar1 t2 in
let (tyT3,nextuvar3,constr3) = recon ctx nextuvar2 t3 in
let newconstr = [(tyT1,TyBool); (tyT2,tyT3)] in
(tyT3, nextuvar3,
List.concat [newconstr; constr1; constr2; constr3]))

```

22.3.11. РЕШЕНИЕ: Правило порождения ограничений для выражений `fix` несложно построить на основе правила типизации `T-FIX` с рис. 11.12.

$$\frac{\Gamma \vdash t_1 : T_1 |_{\mathcal{X}_1} \mathcal{C}_1 \quad X \text{ не встречается в } \mathcal{X}_1, \Gamma \text{ и } t_1}{\Gamma \vdash \text{fix } t_1 : X |_{\mathcal{X}_1 \cup \{X\}} \mathcal{C}_1 \cup \{T_1 = X \rightarrow X\}} \quad (\text{CT-FIX})$$

Это правило реконструирует тип терма t_1 (под именем T_1); проверяет, что T_1 имеет вид $X \rightarrow X$ для некоторой новой переменной X ; и объявляет X типом выражения `fix` t_1 .

В свою очередь, правило порождения ограничений для выражений `letrec` можно построить на основе `CT-FIX` и определения `letrec` как производной формы.

22.4.3. РЕШЕНИЕ:

$\{X = \text{Nat}, Y = X \rightarrow X\}$	$[X \mapsto \text{Nat}, Y \mapsto \text{Nat} \rightarrow \text{Nat}]$
$\{\text{Nat} \rightarrow \text{Nat} = X \rightarrow Y\}$	$[X \mapsto \text{Nat}, Y \mapsto \text{Nat}]$
$\{X \rightarrow Y = Y \rightarrow Z, Z = U \rightarrow W\}$	$[X \mapsto U \rightarrow W, Y \mapsto U \rightarrow W, Z \mapsto U \rightarrow W]$
$\{\text{Nat} = \text{Nat} \rightarrow Y\}$	не унифицируется
$\{Y = \text{Nat} \rightarrow Y\}$	не унифицируется
$\{\}$	\square

22.4.6. РЕШЕНИЕ: Основная структура данных, необходимая для этого упражнения, — это представление подстановок. Ее можно сконструировать по-разному; проще всего будет воспользоваться типом данных `constr` из упражнения 22.3.10; подстановка тогда будет просто множеством ограничений, все

левые части которых — переменные унификации. Если определить функцию `substinty`, которая проводит подстановку типа для одной типовой переменной

```
let substinty tyX tyT tyS =
  let rec f tyS = match tyS with
    TyArr(tyS1,tyS2) → TyArr(f tyS1, f tyS2)
  | TyNat → TyNat
  | TyBool → TyBool
  | TyId(s) → if s=tyX then tyT else TyId(s)
  in f tyS
```

то применение к типу подстановки целиком можно представить так:

```
let applysubst constr tyT =
  List.fold_left
    (fun tyS (TyId(tyX),tyC2) → substinty tyX tyC2 tyS)
    tyT (List.rev constr)
```

Кроме того, функция унификации должна уметь применять подстановку ко всем типам в некотором множестве ограничений:

```
let substinconstr tyX tyT constr =
  List.map
    (fun (tyS1,tyS2) →
      (substinty tyX tyT tyS1, substinty tyX tyT tyS2))
    constr
```

Помимо этого, нужна «проверка на вхождение», которая определяет циклические зависимости:

```
let occursin tyX tyT =
  let rec o tyT = match tyT with
    TyArr(tyT1,tyT2) → o tyT1 || o tyT2
  | TyNat → false
  | TyBool → false
  | TyId(s) → (s=tyX)
  in o tyT
```

Теперь функцию унификации можно записать прямым переводом псевдокода, приведенного на рис. 22.2. Как обычно, она принимает в качестве дополнительных аргументов позицию в файле и строку, которые нужно будет использовать при распечатке сообщений об ошибке, если унификация окажется неудачной.

```
let unify fi ctx msg constr =
  let rec u constr = match constr with
    [] → []
  | (tyS,TyId(tyX)) :: rest →
    if tyS = TyId(tyX) then u rest
    else if occursin tyX tyS then
      error fi (msg ^ ": circular constraints")
    else
      List.append (u (substinconstr tyX tyS rest))
        [(TyId(tyX),tyS)]
```

```

| (TyId(tyX),tyT) :: rest →
  if tyT = TyId(tyX) then u rest
  else if occursin tyX tyT then
    error fi (msg ^ ": circular constraints")
  else
    List.append (u (substinconstr tyX tyT rest))
      [(TyId(tyX),tyT)]
| (TyNat,TyNat) :: rest → u rest
| (TyBool,TyBool) :: rest → u rest
| (TyArr(tyS1,tyS2),TyArr(tyT1,tyT2)) :: rest →
  u ((tyS1,tyT1) :: (tyS2,tyT2) :: rest)
| (tyS,tyT)::rest →
  error fi "Unsolvable constraints"
in
  u constr

```

Эта учебная версия унификации не особенно заботится о том, чтобы сделать сообщения об ошибке полезными и понятными. На практике «объяснение» ошибок типизации может быть одной из самых сложных задач при разработке промышленного компилятора для языка с реконструкцией типов. (См. Wand, 1986).

22.5.6. РЕШЕНИЕ: Расширение алгоритма реконструкции типов на типы записей — задача непростая, хотя и решаемая. Основная сложность состоит в том, что неясно, какие ограничения требуется порождать для проекции записей. Первая, наивная попытка может выглядеть так:

$$\frac{\Gamma \vdash t : T |_{\mathcal{X}} \mathcal{C}}{\Gamma \vdash t.l_i : X |_{\mathcal{X} \cup \{X\}} \mathcal{C} \cup \{T = \{l_i:X\}\}}$$

однако она оказывается неудовлетворительной, поскольку, в сущности, это правило говорит, что поле l_i может получиться исключительно в результате проекции записи, содержащей *только* поле l_i и больше ничего.

Изящное решение было предложено Вандом (Wand, 1987) и впоследствии усовершенствовано Вандом (Wand, 1988, 1989a), Реми (Rémy, 1989, 1990) и др. Введем новую разновидность переменных — *строчные переменные* (row variable). Значениями таких переменных являются не типы, а «строки», состоящие из меток полей и связанных с ними типов. С использованием строчных переменных правило порождения ограничений для проекции поля можно записать в следующем виде:

$$\frac{\Gamma \vdash t_0 : T |_{\mathcal{X}} \mathcal{C}}{\Gamma \vdash t_0.l_i : X |_{\mathcal{X} \cup \{X, \sigma, \rho\}} \mathcal{C} \cup \{T = \{\rho\}, \rho = l_i:X \oplus \sigma\}} \quad (\text{CT-PROJ})$$

где σ и ρ — строчные переменные, а оператор \oplus склеивает две строки (при этом предполагается, что множества их меток полей не пересекаются). То есть, терм $t_0.l_i$ имеет тип X , если t_0 имеет тип записей с полями ρ , а ρ содержит поле $l_i:X$ и еще какие-то поля σ .

Ограничения, порождаемые этим улучшенным алгоритмом, имеют более сложный вид, чем просто множество уравнений, содержащих переменные унификации, как получается при работе исходного алгоритма. Это потому, что но-

вые множества ограничений содержат ассоциативный и коммутативный оператор \oplus . Для нахождения решений таких множеств ограничений требуется простая разновидность *эквациональной унификации* (equational unification).

23.4.3. РЕШЕНИЕ: Вот стандартное решение с использованием вспомогательной функции `append`:

```
append = λX. (fix (λapp:(List X) →(List X) →(List X).
    λl1:List X. λl2:List X.
        if isnil [X] l1 then l2
        else cons [X] (head [X] l1)
            (app (tail [X] l1) l2)));
⇒ append : ∀X. List X →List X → List X

reverse =
λX.
(fix (λrev:(List X) →(List X).
    λl: (List X).
        if isnil [X] l
        then nil [X]
        else append [X] (rev (tail [X] l))
            (cons [X] (head [X] l) (nil [X]))));
⇒ reverse : ∀X. List X →List X
```

23.4.5. РЕШЕНИЕ:

```
and = λb:CBool. λc:CBool.
    λX. λt:X. λf:X. b [X] (c [X] t f) f;
```

23.4.6. РЕШЕНИЕ:

```
iszro = λn:CNat. n [Bool] (λb:Bool. fls) tru;
```

23.4.8. РЕШЕНИЕ:

```
pairNat = λn1:CNat. λn2:CNat.
    λX. λf:CNat →CNat →X. f n1 n2;
fstNat = λp:PairNat. p [CNat] (λn1:CNat. λn2:CNat. n1);
sndNat = λp:PairNat. p [CNat] (λn1:CNat. λn2:CNat. n2);
```

23.4.9. РЕШЕНИЕ:

```
zz = pairNat c0 c0;
f = λp:PairNat. pairNat (sndNat p) (cplus c1 (sndNat p));
prd = λm:CNat. fstNat (m [pairNat] f zz);
```

23.4.10. РЕШЕНИЕ:

```
vpred = λn:CNat. λX. λs:X →X.
    λz:X.
        (n [(X → X) → X]
            (λp:(X →X) → X. λq:(X →X). q (p s))
            (λx:X →X. z))
```

```
(λx:X. x);

▷ vpred : CNat → CNat
```

Я благодарен Майклу Левину, который рассказал мне об этом примере.

23.4.11. РЕШЕНИЕ:

```
head = λX. λdefault:X. λl>List X.
    l [X] (λhd:X. λtl:X. hd) default;
```

23.4.12. РЕШЕНИЕ: Самое сложное в этом упражнении — функция вставки. Решение применяет данный список *l* к функции, которая порождает *два* новых списка, один из которых совпадает с исходным, а другой содержит *e*. Для каждого элемента *hd* в списке *l* (считая справа налево) в функцию передается сам *hd* и пара списков, уже построенных для элементов справа от *hd*. Новая пара строится при сравнении *e* с *hd*: если *e* меньше или равен *hd*, он должен находиться в начале получающегося списка; таким образом, мы строим второй список результата, добавляя *e* к началу *первого* входного списка (того, в котором *e* *ещё* не содержится). С другой стороны, если *e* больше, чем *hd*, то он должен находиться где-то в середине второго списка, и новый список мы порождаем простым добавлением *hd* к уже полученному нами второму списку.

```
insert =
  λX. λleq:X→X→Bool. λl>List X. λe:X.
  let res =
    l [Pair (List X) (List X)]
      (λhd:X. λacc: Pair (List X) (List X).
        let rest = fst [List X] [List X] acc in
        let newrest = cons [X] hd rest in
        let restwithe = snd [List X] [List X] acc in
        let newrestwithe =
          if leq e hd
            then cons [X] e (cons [X] hd rest)
            else cons [X] hd restwithe in
        pair [List X] [List X] newrest newrestwithe)
      (pair [List X] [List X] (nil [X]) (cons [X] e (nil [X])))
    in snd [List X] [List X] res;
```

```
▷ insert : ∀X. (X→X→ Bool) → List X → X → List X
```

Теперь нам нужна функция сравнения для чисел. Так как мы пользуемся числами-примитивами, то писать ее надо с помощью *fix*. (Если бы мы использовали *CNat* вместо *Nat*, можно было бы обойтись вовсе без *fix*).

```
leqnat =
  fix (λf:Nat→Nat→ Bool. λm:Nat. λn:Nat.
    if iszero m then true
    else if iszero n then false
    else f (pred m) (pred n));
```

```
▷ leqnat : Nat → Nat → Bool
```

Наконец, построим функцию сортировки, по очереди вставляя каждый элемент исходного списка в новый список:

```
sort =  $\lambda X. \lambda leq:X \rightarrow X \rightarrow \text{Bool}. \lambda l:\text{List } X.$ 
        $l [ \text{List } X]$ 
        $(\lambda hd:X \lambda rest:\text{List } X. \text{insert } [X] leq rest hd)$ 
        $(\text{nil } [X]);$ 

▷ sort :  $\forall X. (X \rightarrow X \rightarrow \text{Bool}) \rightarrow \text{List } X \rightarrow \text{List } X$ 
```

Чтобы убедиться в том, что сортировка работает правильно, построим неупорядоченный список:

```
l = cons [Nat] 9
      (cons [Nat] 2 (cons [Nat] 6 (cons [Nat] 4 (nil [Nat]))));

отсортируем его
```

```
l = sort [Nat] leqnat l;
```

и считаем содержимое:

```
nth =
 $\lambda X. \lambda \text{default}:X.$ 
    fix  $(\lambda f:(\text{List } X) \rightarrow \text{Nat} \rightarrow X. \lambda l:\text{List } X. \lambda n:\text{Nat}.$ 
        if  $\text{iszero } n$ 
            then head [X] default l
            else f (tail [X] 1) (pred n);

▷ nth :  $\forall X. X \rightarrow \text{List } X \rightarrow \text{Nat} \rightarrow X$ 

nth [Nat] 0 1 0;

▷ 2 : Nat

nth [Nat] 0 1 1;

▷ 4 : Nat

nth [Nat] 0 1 2;

▷ 6 : Nat

nth [Nat] 0 1 3;

▷ 9 : Nat

nth [Nat] 0 1 4;

▷ 0 : Nat
```

Демонстрация того, что на языке Системы F можно реализовать правильно типизированный алгоритм сортировки, была выдающимся достижением Рейнольдса (Reynolds, 1985). Его алгоритм несколько отличался от представленного здесь.

23.5.1. РЕШЕНИЕ: Структура доказательства почти такая же, как в 9.3.9 (см. с. 128). Для правила применения типа Е-ТАРРТАБС требуется дополнительная лемма о подстановке, аналогичная лемме 9.3.8 (см. с. 127).

Если $\Gamma, x, \Delta \vdash t : T$, то $\Gamma, [x \mapsto S]\Delta \vdash [x \mapsto S]t : [x \mapsto S]T$.

Дополнительный контекст Δ здесь требуется, чтобы получить достаточное сильное предположение индукции; если его опустить, то сломается вариант Т-Abs.

23.5.2. РЕШЕНИЕ: Опять же, структура доказательства очень похожа на теорему 9.3.5 — доказательство свойства продвижения для λ_{\rightarrow} . Лемму о канонических формах (9.3.4) следует дополнить одним вариантом

Если v является значением типа $\forall X. T_{12}$, то $v = \lambda X. t_{12}$.

который используется в главном доказательстве в случае с применением типа.

23.6.3. РЕШЕНИЕ: Все части доказательства являются довольно прямолинейными индуктивными и/или вычислительными рассуждениями — кроме самой последней, для которой требуется немного больше изобретательности, чтобы увидеть, как соединить кусочки задачи и получить противоречие. Структуру этого рассуждения предложил Павел Ужичин.

1. Прямолинейная индукция по t , с использованием леммы об инверсии для типизации.
2. Индукцией по количеству внешних абстракций и применений типов покажем, что

Если t имеет вид $\lambda \bar{Y}. (r [\bar{B}])$ для некоторых \bar{Y}, \bar{B} и r (где от r не требуется быть незащищенным), и при этом $\text{erase}(t) = m$ и $\Gamma \vdash t : T$, то имеется некоторый тип s вида $s = \lambda \bar{X}. (u [\bar{A}])$, такой, что $\text{erase}(s) = m$ и $\Gamma \vdash s : T$, и вдобавок терм u незащищен.

В базовом случае нет никаких внешних абстракций и применений типа — т. е., r сам является незащищенным, и требуемое условие выполняется.

В индуктивном случае внешний конструктор r — это либо абстракция типа, либо применение типа. Если это применение типа, скажем, $r_1 [R]$, то мы добавляем R к последовательности \bar{B} и применяем предположение индукции. Если это абстракция типа, скажем, $\lambda Z. r_1$, нужно рассмотреть два подварианта:

- (a) Если последовательность применений \bar{B} пуста, мы можем добавить Z к последовательности абстракций \bar{Y} и применить предположение индукции.
- (b) Если \bar{B} непуста, то мы можем записать t в виде

$$t = \lambda \bar{Y}. ((\lambda Z. r_1) [B_0] [\bar{B}'])$$

где $\bar{B} = B_0 \bar{B}'$. Однако этот терм содержит редекс для правила R-ВЕТА2; редуцируя, получаем терм

$$t' = \lambda \bar{Y}. (([B_0 \mapsto Z]r_1) [\bar{B}'])$$

где $[B_0 \mapsto Z]r_1$ содержит строго меньше внешних абстракций и применений типов, чем r . Более того, теорема о редукции субъекта говорит нам, что тип t' совпадает с типом t . Применение предположения индукции ведет к требуемому результату.

3. Непосредственно следует из леммы об инверсии.
4. Прямолинейное вычисление из пунктов (1), (3) и [дважды] (2).
5. Непосредственно следует из пункта (2) и леммы об инверсии.
6. Индукция по размеру T_1 . В базовом случае, когда T_1 — переменная, эта переменная должна происходить из $\bar{X}_1 \bar{X}_2$, поскольку в противном случае

$$[\bar{X}_1 \bar{X}_2 \mapsto \bar{A}] (\forall \bar{Y}. T_1) = \forall \bar{Y}. W = \forall \bar{Z}. (\forall \bar{Y}. W) \rightarrow ([\bar{X}_1 \mapsto \bar{B}] T_2)$$

а такого случиться не может (слева стрелок нет, а справа есть по меньшей мере одна). Остальные варианты непосредственно следуют из предположения индукции.

7. Предположим, что терм `omega` типизируем, и получим противоречие. Согласно пунктам (1) и (3), существует незащищенный терм $o = s$ и такой, что

$$\begin{aligned} erase(s) &= \lambda x. x \ x \quad erase(u) = \lambda y. y \ y \\ \Gamma \vdash s : U \rightarrow V \quad \Gamma \vdash u : U. \end{aligned}$$

Согласно пункту (2), существуют термы $s' = \lambda \bar{R}. (s_0 [\bar{E}])$ и $u' = \lambda \bar{V}. (u_0 [\bar{F}])$ с незащищенными s_0 и u_0 и

$$\begin{aligned} erase(s_0) &= \lambda x. x \ x \quad erase(u_0) = \lambda y. y \ y \\ \Gamma \vdash s' : U \rightarrow V \quad \Gamma \vdash u' : U. \end{aligned}$$

Поскольку s' имеет функциональный тип, последовательность \bar{R} должна быть пустой. Аналогично, поскольку s_0 и u_0 — незащищенные, они должны начинаться с абстракций, так что \bar{E} и \bar{F} также пусты, и мы имеем

$$\begin{aligned} o' &= s' u' \\ &= s_0 (\lambda \bar{V}. u_0) \\ &= (\lambda x:T_x. w) (\lambda \bar{V}. \lambda y:T_y. v), \end{aligned}$$

где $erase(w) = x \ x$ и $erase(v) = y \ y$. Согласно лемме об инверсии, $U = T_x$ и

$$\Gamma, x:T_x \vdash w : W \quad \Gamma, \bar{V}, y:T_y \vdash v : V$$

Применяя пункт (4) к первому из этих утверждений, имеем либо

(a) $T_x = \forall \bar{X}. X_i$, либо

(b) $T_x = \forall \bar{X}_1 \bar{X}_2 . T_1 \rightarrow T_2$, и для некоторых \bar{A} и \bar{B}

$$[\bar{X}_1 \bar{X}_2 \mapsto \bar{A}]T_1 = [\bar{X}_1 \mapsto \bar{B}](\forall \bar{Z} . T_1 \rightarrow T_2).$$

По пункту (5) T_x должен иметь вторую из этих форм, так что, по пункту (6), самым левым листом типа T_1 будет $X_i \in \bar{X}_1 \bar{X}_2$.

Применяя теперь пункт (4) к типизации $\Gamma, \bar{V}, y:T_y \vdash v : P$, имеем либо

(a) $T_y = \forall \bar{Y} . Y_i$, либо

(b) $T_y = \forall \bar{Y}_1 \bar{Y}_2 . S_1 \rightarrow S_2$, и для некоторых \bar{C} и \bar{D}

$$[\bar{Y}_1 \bar{Y}_2 \mapsto \bar{C}]S_1 = [\bar{Y}_1 \mapsto \bar{D}](\forall \bar{Z} . S_1 \rightarrow S_2).$$

В первом из этих вариантов мы немедленно получаем, что самым левым листом T_y является $Y_i \in \bar{Y}$. Во втором случае мы можем воспользоваться пунктом (6) и, опять же, показать, что самым левым листом T_y будет $Y_i \in \bar{Y}_1 \bar{Y}_2$.

Однако из формы o' и леммы об инверсии мы получаем

$$\begin{aligned} \forall \bar{V} . T_Y \rightarrow V &= T_X \\ &= \forall \bar{X}_1 \bar{X}_2 . T_1 \rightarrow T_2, \end{aligned}$$

так что, в частности, $T_Y = T_1$. Другими словами, самый левый лист типа T_1 совпадает с самым левым листом T_Y . Таким образом, мы имеем $T_X = \forall \bar{X}_1 \bar{X}_2 . (\forall \bar{Y} . S) \rightarrow T_2$, причем одновременно $leftmost-leaf(S) = X_i \in \bar{X}_1 \bar{X}_2$ и $leftmost-leaf(S) = Y_i \in \bar{Y}$. Поскольку переменные $\bar{X}_1 \bar{X}_2$ и \bar{Y} связываются в различных местах, мы вывели противоречие: наше исходное предположение о том, что терм $omega$ является типизируемым — ложно.

23.7.1. РЕШЕНИЕ:

```
let r = λX. ref (λx:X. x) in
(r[Nat] := (λx:Nat. succ x);
 (! (r[Bool]))) true);
```

24.1.1. РЕШЕНИЕ: Пакет p6 предоставляет пользователю константу a и функцию f , но единственная операция, разрешенная типами этих компонентов, состоит в том, чтобы применить f к a сколько-то раз, а затем выбросить результат. Пакет p7 позволяет пользователю создавать значения типа X , но ничего сделать с этими значениями мы не можем. В пакете p8 оба компонента можно использовать, но ничего не спрятано — с тем же успехом мы можем просто отбросить упаковку.

24.2.1. РЕШЕНИЕ:

```
stackADT =
{*List Nat,
 {new = nil [Nat],
 push = λn:Nat. λs>List Nat. cons [Nat] n s,
 top = λs>List Nat. head [Nat] s,
```

```

pop = λs>List Nat. tail [Nat] s,
isempty = isnil [Nat]}
as {∃Stack, {new: Stack, push: Nat → Stack →
Stack, top:Stack → Nat,
pop: Stack → Stack, isempty: Stack → Bool}};

▷ stackADT : {∃Stack,
{new: Stack, push: Nat → Stack → Stack, top:Stack →
Nat,
pop: Stack → Stack, isempty: Stack → Bool}}

let {Stack,stack} = stackADT in
stack.top (stack.push 5 (stack.push 3 stack.new));

▷ 5 : Nat

```

24.2.2. РЕШЕНИЕ:

```

counterADT =
{*Ref Nat,
{new = λ_:Unit. ref 1,
get = λr:Ref Nat. !r,
inc = λr:Ref Nat. r := succ(!r)}}
as {∃Counter,
{new: Unit → Counter, get:Counter →
Nat, inc:Counter → Unit}};

▷ counterADT : {∃Counter,
{new: Unit → Counter, get:Counter → Nat,
inc:Counter → Unit}}

```

24.2.3. РЕШЕНИЕ:

```

FlipFlop = {∃X, {state:X, methods: {read: X →
Bool, toggle: X → X,
reset: X → X}}};

f = {*Counter,
{state = zeroCounter,
methods = {read = λs:Counter. iseven (sendget s),
toggle = λs.Counter. sendinc s,
reset = λs:Counter. zeroCounter}}}
as FlipFlop;

▷ f : FlipFlop

```

24.2.4. РЕШЕНИЕ:

```

c = {*Ref Nat,
{state = ref 5,
methods = {get = λx:Ref Nat. !x,
inc = λx:Ref Nat. (x := succ(!x); x)}}}
as Counter;

```

24.2.5. РЕШЕНИЕ: Такой тип позволит нам *реализовать* объекты-множества с методами `union`, но не позволит их *использовать*. Чтобы вызвать метод `union` у такого объекта, нужно передать ему два значения одного и того же типа представления `X`. Однако эти значения не могут происходить из двух разных объектов-множеств, поскольку, чтобы получить состояния обоих, нам пришлось бы их оба открыть, и при этом оказались бы связаны две различные типовые переменные; состояние второго множества нельзя было бы передать в операцию `union` первого. (Это не просто упрямство программы проверки типов: нетрудно заметить, что передавать конкретное представление одного множества в операцию `union` другого было бы неправильно, поскольку представление второго множества может в общем случае сколь угодно отличаться от представления первого.) Таким образом, эта версия типа `NatSet` позволяет нам только вычислять объединение множества с самим собой!

24.3.2. РЕШЕНИЕ: По меньшей мере, требуется показать, что правила типизации и вычисления для экзистенциальных типов сохраняются при переводе — т. е. если обозначить функцию, которая проводит перевод, через $\llbracket _ \rrbracket$, то нужно показать, что из $\Gamma \vdash t : T$ следует $\llbracket \Gamma \rrbracket \vdash \llbracket t \rrbracket : \llbracket T \rrbracket$, и что из $t \rightarrow^* t'$ следует $\llbracket t \rrbracket \rightarrow^* \llbracket t' \rrbracket$. Эти свойства легко проверить. Можно было бы также надеяться, что верно и обратное — что некорректно типизированные термы языка, включающего экзистенциальные типы, всегда переводятся в некорректно типизированные термы, а тупиковые термы в тупиковые; к сожалению, эти свойства не выполняются. Например, при переводе неверно типизированному (и тупиковому) терму $\{*\text{Nat}, 0\} \text{ as } \{\exists, x\}$ `[Bool]` соответствует правильно типизированный (и не тупиковый) терм.

24.3.3. РЕШЕНИЕ: Насколько мне известно, такое преобразование нигде не описано. По-видимому, оно должно быть возможно, но трансформация не будет простым синтаксическим сахаром — ее придется применять ко всей программе целиком.

25.2.1. РЕШЕНИЕ: Сдвиг типа `T` на d позиций при отсечении c , что записывается как $\uparrow_c^d(T)$, определяется следующим образом:

$$\begin{aligned}\uparrow_c^d(k) &= \begin{cases} k & \text{если } k < c \\ k + d & \text{если } k \geq c \end{cases} \\ \uparrow_c^d(T_1 \rightarrow T_2) &= \uparrow_c^d(T_1) \rightarrow \uparrow_c^d(T_2) \\ \uparrow_c^d(\forall. T_1) &= \forall. \uparrow_{c+1}^d(T_1) \\ \uparrow_c^d(\{\exists, T_1\}) &= \{\exists, \uparrow_{c+1}^d(T_1)\}\end{aligned}$$

Запись $\uparrow_0^d(T)$ обозначает сдвиг на d позиций всех переменных в типе `T`, т. е. $\uparrow^d(T)$.

25.4.1. РЕШЕНИЕ: Он освобождает место для типовой переменной `X`. Предполагается, что результат подстановки `v12` в терм `t2` правильно определен относительно контекста вида Γ, X , тогда как исходное значение `v12` определено относительно просто Γ .

26.2.3. РЕШЕНИЕ: Кодирование объектов по Абади, Карделли и Вишванатану (Abadi, Cardelli, and Viswanathan, 1996) — одна из задач, в которых требуетсѧ полное правило $F_{<:}$. Такое кодирование также обсуждается в (Abadi and Cardelli, 1996).

26.3.4. РЕШЕНИЕ:

```
spluszz = λn:SZero. λm:SZero.
          λX. λS<:X. λZ<:X. λs:X →S. λz:Z.
          n [X] [S] [Z] s (m [X] [S] [Z] s z);

spluspn = λn:SPos. λm:SNat.
          λX. λS<:X. λZ<:X. λs:X →S. λz:Z.
          n [X] [S] [X] s (m [X] [S] [Z] s z);

▷ spluspn : SPos → Nat → SPos
```

26.3.5. РЕШЕНИЕ:

```
SBool = ∀X. ∀T<:X. ∀F<:X. T →F →X;
STrue = ∀X. ∀T<:X. ∀F<:X. T →F →T;
SFalse = ∀X. ∀T<:X. ∀F<:X. T →F →F;
tru = λX. λT<:X. λF<:X. λt:T. λf:F. t;

▷ tru : STrue

fls = λX. λT<:X. λF<:X. λt:T. λf:F. f;

▷ fls : SFalse

notft = λb:SFalse. λX. λT<:X. λF<:X. λt:T. λf:F. b[X][F][T] f t;

▷ notft : SFalse → STrue

nottf = λb:STrue. λX. λT<:X. λF<:X. λt:T. λf:F. b[X][F][T] f t;

▷ nottf : STrue → SFalse
```

26.4.3. РЕШЕНИЕ: В вариантах с абстракцией и абстракцией типа в частях (1) и (2), а также в варианте с квантором в частях (3) и (4).

26.4.5. РЕШЕНИЕ: В части (1) проводится индукция по деревьям вывода подтипования. Все варианты получаются либо непосредственно (S-REFL, S-TOP), либо несложным применением предположения индукции (S-TRANS, S-ARROW, S-ALL), за исключением S-TVAR, где дела обстоят интереснее. Предположим, что последнее правило при выводе $\Gamma, X<:Q, \Delta \vdash S <: T$ — экземпляр правила S-TVAR, т. е. S представляет собой некую переменную Y , а T является верхней границей для этой переменной в контексте. Нужно рассмотреть два случая. Если X и Y — разные переменные, то предположение $Y<:T$ имеется также в контексте $\Gamma, X<:P, \Delta$, и тогда результат следует непосредственно. Если же $X = Y$, то $T = Q$; чтобы завершить доказательство, нужно показать, что $\Gamma, X<:P, \Delta \vdash X <: Q$. По правилу S-TVAR имеем $\Gamma, X<:P, \Delta \vdash X <: P$. Более того, по предположению, $\Gamma \vdash P <: Q$, и по лемме об ослаблении (26.4.2), $\Gamma, X<:P, \Delta \vdash P <: Q$. Объединение этих двух выводов через S-TRANS дает нам требуемый результат.

Часть (2) представляет собой несложную индукцию по деревьям вывода типов, с использованием части (1) для предпосылки о подтиповании в варианте с применением типа.

26.4.11. РЕШЕНИЕ: Все доказательства представляют собой прямолинейные индукции по деревьям вывода подтиповирования. Мы приводим только первое из них, рассматривая варианты последнего правила в выводе. Варианты S-REFL и S-TOP следуют непосредственно. Вариант S-TVAR возникнуть не может (левая часть заключения S-TVAR — это всегда переменная, а не функциональный тип); точно так же невозможно правило S-ALL. Если последнее правило — экземпляр S-ARROW, то поддеревья дают требуемые результаты. Наконец, предположим, что последнее правило — экземпляр S-TRANS, т. е. мы имеем $\Gamma \vdash S_1 \rightarrow S_2 <: U$ и $\Gamma \vdash U <: T$ для некоторого U . По предположению индукции либо U равен Top (тогда T тоже равен Top по пункту (4) данного упражнения, и требуемый результат получен), либо U имеет вид $U_1 \rightarrow U_2$, причем $\Gamma \vdash U_1 <: S_1$ и $\Gamma \vdash S_2 <: U_2$. Во втором случае мы применяем предположение индукции ко второму подвыводу исходного S-TRANS и получаем, что либо $T = \text{Top}$ (и утверждение доказано), либо T имеет вид $T_1 \rightarrow T_2$, причем $\Gamma \vdash T_1 <: U_1$ и $\Gamma \vdash U_2 <: T_2$. Два применения транзитивности дают нам $\Gamma \vdash T_1 <: S_1$ и $\Gamma \vdash S_2 <: T_2$, откуда требуемый результат следует по правилу S-ARROW.

26.5.1. РЕШЕНИЕ:

$$\frac{\Gamma \vdash S_1 <: T_1 \quad \Gamma, \exists X : S_1 \vdash S_2 <: T_2}{\Gamma \vdash \{\exists X : S_1, S_2\} <: \{\exists X : T_1, T_2\}} \quad (\text{S-SOME})$$

26.5.2. РЕШЕНИЕ: Без подтипов имеется всего четыре типа:

```
{*Nat, {a=5,b=7}} as {EX, {a:Nat,b:Nat}};
{*Nat, {a=5,b=7}} as {EX, {a:X,b:Nat}};
{*Nat, {a=5,b=7}} as {EX, {a:Nat,b:X}};
{*Nat, {a=5,b=7}} as {EX, {a:X,b:X}};
```

С подтипами и ограниченной квантификацией добавляется множество других — например,

```
{*Nat, {a=5,b=7}} as {EX, {a:Nat}};
{*Nat, {a=5,b=7}} as {EX, {b:X}};
{*Nat, {a=5,b=7}} as {EX, {a:Top,b:X}};
{*Nat, {a=5,b=7}} as {EX, Top};
{*Nat, {a=5,b=7}} as {\exists X : Nat, {a:X,b:X}};
{*Nat, {a=5,b=7}} as {\exists X : Nat, {a:Top,b:X}};
```

26.5.3. РЕШЕНИЕ: Один из способов достичь желаемого — вложить АТД счетчика со сбросом *внутрь* АТД обычного счетчика:

```
counterADT =
  {*Nat,
   {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i),
    rcADT =
      {*Nat,
       {new = 1, get = λi:Nat. i, inc = λi:Nat. succ(i),
        reset = λi:Nat. 1}}
     as {\exists ResetCounter <: Nat,
          {new: ResetCounter, get: ResetCounter → Nat,
```

```

        inc: ResetCounter → ResetCounter ,
        reset: ResetCounter → ResetCounter}} } }

as {ΞCounter ,
  {new: Counter , get: Counter → Nat , inc: Counter →
Counter ,
  rcADT:
    {ΞResetCounter<: Counter ,
     {new: ResetCounter , get: ResetCounter → Nat ,
      inc: ResetCounter → ResetCounter ,
      reset: ResetCounter → ResetCounter}}}};

▷ counterADT : {ΞCounter ,
  {new: Counter , get: Counter → Nat , inc: Counter →
Counter ,
  rcADT:{ΞResetCounter<: Counter ,
  {new: ResetCounter , get: ResetCounter →
Nat ,
  inc: ResetCounter → ResetCounter ,
  reset: ResetCounter →
ResetCounter}}}}}

```

Когда эти пакеты открываются, получается, что контекст, в котором происходит проверка типов в оставшейся части программы, содержит связывания вида Counter<:Top, counter:{...}, ResetCounter<:Counter, resetCounter:{...}:

```

let {Counter, counter} = counterADT in
let {ResetCounter, resetCounter} = counter.rcADT in
counter.get
(counter.inc
(resetCounter.reset (resetCounter.inc resetCounter.new)));

```

▷ 2 : Nat

26.5.4. РЕШЕНИЕ: Требуется только добавить границы в очевидные места в кодирование по §24.3. На уровне типов получаем:

$$\{ \exists X <: S, T \} \stackrel{\text{def}}{=} \forall Y. (\forall X <: S. T \rightarrow Y) \rightarrow Y$$

Отсюда непосредственно следуют изменения на уровне термов.

27.0.1. РЕШЕНИЕ: Вот один из способов:

```

setCounterClass =
  λM<: SetCounter . λR<: CounterRep .
  λself: Ref(R → M) .
  λr: R .
  {get = λ_:Unit. !(r.x) ,
   set = λi:Nat. r.x:=i ,
   inc = λ_:Unit. (!self r).set (succ((!self r).get unit))};

▷ setCounterClass : ∀M<: SetCounter .
  ∀R<: CounterRep .
  (Ref (R → M)) → R → SetCounter

```

```

instrCounterClass =
  λM<: InstrCounter.
  λR<: InstrCounterRep.
  λself: Ref(R → M).
  λr: R.
  let super = setCounterClass [M] [R] self in
    {get = (super r).get,
     set = λi:Nat. (r.a := succ(! (r.a)); (super r).set i),
     inc = (super r).inc,
     accesses = λ_:Unit. !(r.a)};

▷ instrCounterClass : ∀M<:InstrCounter.
  ∀R<:InstrCounterRep.
  (Ref (R → M)) → R → InstrCounter

newInstrCounter =
  let m = ref (λr:InstrCounterRep. error as InstrCounter) in
  let m' =
    instrCounterClass [InstrCounter] [InstrCounterRep] m in
  (m := m';
   λ_:Unit. let r = {x=ref 1 a=ref 0} in m' r);

▷ newInstrCounter : Unit → InstrCounter

```

28.2.3. РЕШЕНИЕ: В варианте Т-TABS требуется тривиальным образом использовать S-REFL, чтобы снабдить правило S-ALL дополнительной предпосылкой. В варианте Т-TAPP лемма об инверсии подтиповирования (для полной $F_{<:}$) говорит нам, что $N_1 = \forall X <: N_{11}. N_{12}$, причем $\Gamma \vdash T_{11} <: N_{11}$ и $\Gamma, X <: T_{11} \vdash N_{12} <: T_{12}$. Благодаря транзитивности, имеем $\Gamma \vdash T_2 <: T_{11}$, и поэтому мы имеем право применить TA-TAPP и получить $\Gamma \vdash t_1 [T_2] : [X \mapsto T_2]N_{12}$. Завершаем доказательство, как и раньше, с помощью леммы о сохранении подтиповирования при подстановке (лемма 26.4.8) получая $\Gamma \vdash [X \mapsto T_2]N_{12} <: [X \mapsto T_2]T_{12} = T$.

28.5.1. РЕШЕНИЕ: Для полной $F_{<:}$ неверна теорема 28.3.5 (а именно, вариант с правилом S-ALL).

28.5.6. РЕШЕНИЕ: Прежде всего, заметим, что нельзя смешивать ограниченные и неограниченные кванторы: должно быть отдельное правило подтиповирования для сравнения двух ограниченных кванторов и отдельное правило для сравнения двух неограниченных кванторов. Иначе мы снова окажемся на том же месте, с которого начинали!

Что касается частей (1) и (2), подробности можно найти у Катияра и Санкарса ([Katiyar and Sankar, 1992](#)). В части (3) ответ отрицательный: при добавлении типов-записей с подтиповированием в ширину система снова оказывается неразрешимой. Проблема в том, что пустой тип записей играет роль своего рода наибольшего типа (для записей), и с его помощью можно заставить программу проверки типов зациклиться, используя модифицированную версию примера Гелли. Если $T = \forall X <: \{\}. \neg\{a: \forall Y <: X. \neg Y\}$, то при подаче на вход $X_0 <: \{a: T\} \vdash X_0 <: \{a: \forall X_1 <: X_0. \neg X_1\}$ проверка типов никогда не закончит-

$R(\forall Y <: S.T)$	=	$\begin{cases} \forall Y <: S. R(T) \\ \text{если } X \notin FV(S) \\ \text{Top} \\ \text{если } X \in FV(S) \\ \{\exists Y <: S, R(T)\} \\ \text{если } X \notin FV(S) \\ \text{Top} \\ \text{если } X \in FV(S) \\ L(S) \rightarrow R(T) \\ \text{если } L(S) \neq \text{неудача} \\ \text{Top} \\ \text{если } L(S) = \text{неудача} \end{cases}$	$L(\forall Y <: S.T)$	=	$\begin{cases} \forall Y <: S. L(T) \\ \text{если } L(T) \\ \text{и } X \notin F \\ \text{иначе неудача} \\ \{\exists Y <: S. L(T)\} \\ \text{если } L(T) \\ \text{и } X \notin F \\ \text{иначе неудача} \\ R(S) \rightarrow L(T) \\ \text{если } L(T) \\ \text{неудача} \\ \text{если } L(T) \end{cases}$
$R(\{\exists Y <: S.T\})$	=	$\begin{cases} \exists Y <: S, R(T) \\ \text{если } X \in FV(S) \\ L(S) \rightarrow R(T) \\ \text{если } L(S) \neq \text{неудача} \\ \text{Top} \\ \text{если } L(S) = \text{неудача} \end{cases}$	$L(\{\exists Y <: S.T\})$	=	$\begin{cases} \exists Y <: S. L(T) \\ \text{если } L(T) \\ \text{и } X \notin F \\ \text{иначе неудача} \\ R(S) \rightarrow L(T) \\ \text{если } L(T) \\ \text{неудача} \\ \text{если } L(T) \end{cases}$
$R(S \rightarrow T)$	=	$\begin{cases} T, \text{ где } X <: T \in \Gamma \\ \text{Top} \\ \text{если } L(S) = \text{неудача} \end{cases}$	$L(S \rightarrow T)$	=	$\begin{cases} \text{неудача} \\ \text{Y, когда } Y \neq X \\ \text{Top} \end{cases}$
$R(X)$	=	T	$L(X)$	=	неудача
$R(Y)$	=	$Y, \text{ когда } Y \neq X$	$L(Y)$	=	$Y, \text{ когда } Y \neq X$
$R(\text{Top})$	=	Top	$L(\text{Top})$	=	Top

Рис. A.2. Наименьший надтип и наибольший подтип данного типа, не содержащие X

ся.

При подготовке этого примера мне помог Мартин Хоффман. То же самое наблюдение высказано в (Katiyar and Sankar, 1992).

28.6.3. РЕШЕНИЕ:

1. Я вижу 9 общих подтипов:

$$\begin{array}{lll} \forall X <: Y' \rightarrow Z. \quad Y \rightarrow Z' & \forall X <: Y' \rightarrow Z. \quad \text{Top} \rightarrow Z' & \forall X <: Y' \rightarrow Z. \quad X \\ \forall X <: Y' \rightarrow \text{Top}. \quad Y \rightarrow Z' & \forall X <: Y' \rightarrow \text{Top}. \quad \text{Top} \rightarrow Z' & \forall X <: Y' \rightarrow \text{Top}. \quad X \\ \forall X <: \text{Top}. \quad Y \rightarrow Z' & \forall X <: \text{Top}. \quad \text{Top} \rightarrow Z' & \forall X <: \text{Top}. \quad X \end{array}$$

2. Типы $\forall X <: Y' \rightarrow Z. \quad Y \rightarrow Z'$ и $\forall X <: Y' \rightarrow Z. \quad X$ являются нижними гранями для S и T, но у них нет общего надтипа, который был бы при этом подтиром S и T.

3. Рассмотрите $S \rightarrow \text{Top}$ и $T \rightarrow \text{Top}$. (Или $\forall X <: Y' \rightarrow Z. \quad Y \rightarrow Z'$ и $\forall X <: Y' \rightarrow Z. \quad X$.)

28.7.1. РЕШЕНИЕ: Функции $R_{x,\Gamma}$ и $L_{x,\Gamma}$, переводящие каждый тип в его наименьший надтип, не содержащий X, и в наибольший подтип, не содержащий X, соответственно, определены на рис. A.2. (Чтобы облегчить чтение, мы опускаем индексы X и Г.) У этих двух определений различные дополнительные условия, поскольку, когда требуется вычислить L, нужно проверять, определена ли она (что записывается в виде $L(T) \neq \text{неудача}$); R же определена всегда, поскольку в системе имеется тип Top. Корректность этих определений доказывается в (Ghelli and Pierce, 1998).

28.7.2. РЕШЕНИЕ: Один из простых способов показать неразрешимость полных ограниченных экзистенциальных типов (Гелли и Пирс, Ghelli and Pierce,

1998) заключается в определении функции перевода $\llbracket _ \rrbracket$, которая отображает задачи проверки подтипования в полной $F_<$: в задачи проверки подтипования в системе, содержащей только экзистенциальные типы, так что $\Gamma \vdash S <: T$ доказуемо в $F_<$, тогда и только тогда, когда в системе с экзистенциальными типами доказуемо $\llbracket \Gamma \vdash S <: T \rrbracket$. Такое кодирование на типах можно определить уравнениями

$$\begin{array}{rcl} \llbracket X \rrbracket & = & X \\ \llbracket \forall X <: T_1 . T_2 \rrbracket & = & \neg \{ \exists X <: T_1 , \neg \llbracket T_2 \rrbracket \} \quad \llbracket T_1 \rightarrow T_2 \rrbracket & = & \llbracket T_1 \rrbracket \rightarrow \llbracket T_2 \rrbracket \end{array}$$

где $\neg S = \forall X <: S . X$. Мы распространяем это кодирование на контексты с помощью $\llbracket X_1 <: T_1, \dots, X_n <: T_n \rrbracket = X_1 <: \llbracket T_1 \rrbracket, \dots, X_n <: \llbracket T_n \rrbracket$ и на утверждения о подтиповании: $\llbracket \Gamma \vdash S <: T \rrbracket = \llbracket \Gamma \rrbracket \vdash \llbracket S \rrbracket <: \llbracket T \rrbracket$.

29.1.1. РЕШЕНИЕ: $\forall X . X \rightarrow X$ — простой тип; его элементом будет, например, $\lambda X . \lambda x : X . x$. Такие термы являются полиморфными функциями, которые, будучи конкретизированы типом T , дают функцию из T в T . Напротив, $\lambda X . X \rightarrow X$ — оператор над типами, т. е. функция, которая, будучи применена к типу T , выдает простой тип $T \rightarrow T$ функций из T в T .

Другими словами, $\forall X . X \rightarrow X$ — тип, *элементами* которого являются функции на уровне термов, переводящие типы в термы; конкретизация такой функции (для получения которой нужно применить ее к типу, что записывается в виде $t \ [T]$) дает нам *элемент* функционального типа $T \rightarrow T$. С другой стороны, $\lambda X . X \rightarrow X$ сам по себе является функцией (из типов в типы); его конкретизация типом T (что записывается как $(\lambda X . X \rightarrow X) T$) дает *сам* тип $T \rightarrow T$, а не один из его элементов.

Например, если fn имеет тип $\forall X . X \rightarrow X$, а $Op = \lambda X . X \rightarrow X$, то $fn \ [T] : T \rightarrow T = Op \ T$.

29.1.2. РЕШЕНИЕ: $\text{Nat} \rightarrow \text{Nat}$ представляет собой (простой) тип функций, а не функцию на уровне типов.

30.3.3. РЕШЕНИЕ: Лемма 30.3.1 используется в вариантах T-Abs, T-TAPP и T-EQ. Лемма 30.3.2 используется в варианте T-VAR.

30.3.8. РЕШЕНИЕ: Индукция по сумме размеров данных деревьев вывода, с разбором вариантов финальных правил в обоих из них. Если какой-то из выводов заканчивается на QR-REFL, то второй является требуемым результатом. Если какой-то из выводов заканчивается на QR-ABS, QR-ARROW или QR-ALL, то, исходя из формы этих правил, оба вывода должны заканчиваться одним и тем же правилом, и результат получается простым применением предположения индукции. Если оба вывода заканчиваются на QR-APP, то результат, опять же, попросту следует из предположения индукции. Оставшиеся случаи более интересны.

Если оба вывода заканчиваются на QR-APPABS, то мы имеем

$$S = (\lambda X :: K_{11} . S_{12}) \ S_2 \quad T = [X \mapsto T_2] T_{12} \quad U = [X \mapsto U_2] U_{12},$$

причем

$$S_{12} \Rightarrow T_{12} \quad S_2 \Rightarrow T_2 \quad S_{12} \Rightarrow U_{12} \quad S_2 \Rightarrow U_2.$$

Согласно предположению индукции, существуют такие типы V_{12} и V_2 , что

$$T_{12} \Rightarrow V_{12} \quad T_2 \Rightarrow V_2 \quad U_{12} \Rightarrow V_{12} \quad U_2 \Rightarrow V_2.$$

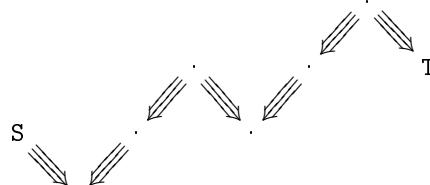
Дважды применяя лемму 30.3.7, получаем $[X \mapsto T_2]T_{12} \Rightarrow [X \mapsto V_2]V_{12}$ и $[X \mapsto U_2]U_{12} \Rightarrow [X \mapsto V_2]V_{12}$ — то есть, $T \Rightarrow V$ и $U \Rightarrow V$.

Допустим, наконец, что один из выводов (скажем, первый) заканчивается на QR-APP, а другой — на QR-APPABS. В этом случае имеем

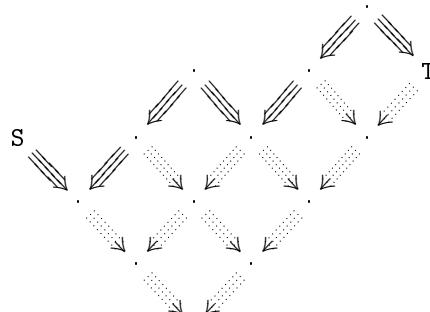
$$S = (\lambda X :: K_{11}. S_{12}) \quad S_2 \quad T = (\lambda X :: K_{11}. T'_{12}) \quad T'_2 \quad U = [X \mapsto U_2]U_{12},$$

где снова $S_{12} \Rightarrow T_{12}$, $S_2 \Rightarrow T_2$, $S_{12} \Rightarrow U_{12}$ и $S_2 \Rightarrow U_2$. Снова, согласно предположению индукции, имеются типы V_{12} и V_2 , такие, что $T_{12} \Rightarrow V_{12}$, $T_2 \Rightarrow V_2$, $U_{12} \Rightarrow V_{12}$ и $U_2 \Rightarrow V_2$. Применение правила QR-APPABS к первому и второму из этих утверждений, а леммы 30.3.7 — к третьему и четвертому, дает нам $T \Rightarrow V$ и $U \Rightarrow V$.

30.3.10. Решение: Прежде всего заметим, что любой вывод утверждения $S \Leftarrow\Rightarrow^* T$ можно перестроить так, чтобы ни симметрия, ни транзитивность не использовалась в подвыводах экземпляров правила симметрии — то есть мы можем перейти от S к T через последовательность шагов, склеенных правилом QR-TRANS, где каждый шаг состоит из одношаговой редукции, за которой, возможно, следует одно применение правила симметрии. Эту последовательность можно изобразить так:



(стрелки, направленные справа налево, соответствуют редукциям, которые заканчиваются применением симметрии, а стрелки слева направо — редукциям без симметрии). Теперь мы можем в цикле при помощи леммы 30.3.8 добавлять маленькие ромбы к низу диаграммы, пока не доберемся до общего результата редукции S и T .



То же самое рассуждение можно представить и в стандартной индуктивной форме, без использования картинок, но это только сделает его сложнее для понимания, не добавляя убедительности.

30.3.17. РЕШЕНИЕ: При добавлении первого странного правила сломается свойство продвижения; свойство сохранения останется верным. Второе правило сломает как продвижение, так и сохранение.

30.3.20. РЕШЕНИЕ: Сравните свое решение с исходными текстами интерпретатора `fomega`.

30.5.1. РЕШЕНИЕ: Вместо семейства типов `List n` у нас будет *параметрическое* семейство типов `List T n` со следующими операциями:

```
nil    : ∀X. List X 0
cons   : ∀X. ∀n:Nat. X → List X (succ n)
hd     : ∀X. ∀n:Nat. List X (succ n) → X
tl     : ∀X. ∀n:Nat. List X (succ n) → List X n
```

31.2.1. РЕШЕНИЕ:

$\Gamma \vdash A$	$<: \text{Id } B$	Да
$\Gamma \vdash \text{Id } A$	$<: B$	Да
$\Gamma \vdash \lambda X. X$	$<: \lambda X. \text{Top}$	Да
$\Gamma \vdash \lambda X. \forall Y <: X. Y$	$<: \lambda X. \forall Y <: \text{Top}. Y$	Нет
$\Gamma \vdash \lambda X. \forall Y <: X. Y$	$<: \lambda X. \forall Y <: X. X$	Да
$\Gamma \vdash F B$	$<: B$	Да
$\Gamma \vdash B$	$<: F B$	Нет
$\Gamma \vdash F B$	$<: F B$	Да
$\Gamma \vdash \forall F <: (\lambda Y. \text{Top} \rightarrow Y). F A$	$<: \forall F <: (\lambda Y. \text{Top} \rightarrow Y). \text{Top} \rightarrow B$	Да
$\Gamma \vdash \forall F <: (\lambda Y. \text{Top} \rightarrow Y). F A$	$<: \forall F <: (\lambda Y. \text{Top} \rightarrow Y). F B$	Нет
$\Gamma \vdash \text{Top} [* \Rightarrow *]$	$<: \text{Top} [* \Rightarrow * \Rightarrow *]$	Нет

32.5.1. РЕШЕНИЕ: Главное, что требуется заметить — `Object M` является экзистенциальным типом: `Object` служит сокращенной записью оператора

```
 $\lambda M :: * \Rightarrow *. \{ \exists X, \{ \text{state}: X, \text{methods}: M X \} \}$ 
```

При применении этого оператора к `M` получается редекс, сводящийся к экзистенциальному типу

```
{ $\exists X, \{ \text{state}: X, \text{methods}: M X \} \}.$ 
```

Заметим, что в этом преобразовании никак не используется включение — и, следовательно, нет потери информации.

32.5.2. РЕШЕНИЕ:

```
sendget =
   $\lambda M <: \text{CounterM}. \lambda o: \text{Object } M.$ 
    let {X, b} = o in b.methods.get(b.state);

sendreset =
   $\lambda M <: \text{ResetCounterM}. \lambda o: \text{Object } M.$ 
    let {X, b} = o in
```

```
{*X,
  {state = b.methods.reset(b.state),
   methods = b.methods}} as Object M;
```

32.7.2. РЕШЕНИЕ: Свойство минимальной типизации не работает в исчислении, как мы его определили. Рассмотрим терм

```
{#x={a=5,b=7}}.
```

Ему можно присвоить любой из типов `{#x:{a:Nat}}` и `{#x:{a:Nat,b:Nat}}`, но эти два типа несравнимы. Один из разумных способов устранения этого затруднения — явно указывать при каждом инвариантном поле в терме-записи его требуемый тип. По существу, это заставляет программиста сделать выбор между двумя указанными типами.

32.9.1. РЕШЕНИЕ:

```
MyCounterM =
  λR. {get: R → Nat, set: R → Nat → R, inc: R →
R, accesses: R → Nat,
      backup: R → R, reset: R → R};

MyCounterR = {#x:Nat, #count:Nat, #old:Nat};

myCounterClass =
  λR<:MyCounterR.
  λself: Unit → MyCounterM R.
  λ_:Unit.
  let super = instrCounterClass [R] self unit in
  {get = super.get,
   set = super.set,
   inc = super.inc,
   accesses = super.accesses,
   reset = λs:R. s ← x = s.old,
   backup = λs:R. s ← old = s.x}
  as MyCounterM R;

mc = {*MyCounterR,
       {state = {#x=0, #count=0, #old=0},
        methods = fix (myCounterClass [MyCounterR]) unit}}
  as Object MyCounterM;

sendget [MyCounterM]
(sendreset [MyCounterM] (sendinc [MyCounterM] mc));
```

— Дорогой Уотсон,
проанализируйте факты, —
сказал он с легким
раздражением, — Вы знаете мой
метод. Примените его, будет
интересно сравнить результаты.

А. Конан-Дойль, *Знак четырех*
(1890), пер. М. Литвиновой

Безупречное изложение этой
идеи можно оставить в качестве
упражнения для читателя.
Я прибегаю здесь к любимому
триюку математиков, который
позволяет им перескакивать
через болотистые участки своих
теорий.

У. в. О. Куайн (1987)

Приложение В

Принятые обозначения

В. Имена метапеременных

В ТЕКСТЕ	В КОДЕ НА ML	ЗНАЧЕНИЕ
p, q, r, s, t, u	s, t	термы
x, y, z	x, y	термовые переменные
v, w	v, w	значения
nv	nv	числовые значения
l, j, k	l	метки полей в записях, метки вариантов состояния памяти
μ	$store$	
M, N, P, Q, S, T, U, V	tyS, tyT	типы
A, B, C	tyA, tyB	базовые типы
Σ		структуры типизации памяти
X, Y, Z	tyX, tyY	типовыe переменные
K, L	kK, kL	виды
σ		подстановки
Γ, Δ	ctx	контексты
\mathcal{J}		произвольные утверждения
\mathcal{D}		деревья вывода типов
\mathcal{C}		деревья вывода подтиповирования
	fi	информация о позиции в файле
i, j, k, l		числовые индексы

B. Имена правил

ПРЕФИКС	ЗНАЧЕНИЕ
В-	вычисление с большим шагом
СТ-	типизация с ограничениями
Е-	вычисление
К-	присвоение видов
М-	сопоставление с образцом
Р-	типизация образцов
Q-	эквивалентность типов
QR-	параллельная редукция типов
S-	подтипирование
SA-	алгоритмическое подтипирование
T-	типизация
ТА-	алгоритмическая типизация
ХА-	выявление

B. Соглашения по именам и индексам

При подборе имен метапеременных, числовых индексов и штрихов в книге мы руководствовались следующими принципами:

1. В определениях синтаксиса для всех термов используется простая метапеременная t , для типов — T , для значений — v и т. д.
2. В правилах типизации главный терм (тип которого вычисляется) всегда называется t , его подтермы называются t_1, t_2 и т. д. (Иногда, например, в правилах редукции, требуются имена для подтермов подтермов; для них мы используем t_{11}, t_{12} и т. д.)
3. В правилах вычисления весь редуцируемый терм называется t , а терм-результат редукции называется t' .
4. Тип терма t называется T . (Аналогично, типом t_1 будет T_1 и т. д.)
5. Те же соглашения используются при формулировке и доказательстве теорем, но t иногда заменяется на s (а T — на S или R и т. п.) во избежание смешения имен между определениями и теоремами.

В некоторых случаях выполнить все эти правила одновременно оказывается невозможным. Тогда правила, указанные раньше, имеют более высокий приоритет. (Например, в правиле $T\text{-PROJ1}$ на рис. 11.5 правило 4 ослабляется: типом подтерма t_1 служит $T_1 \times T_2$.) В очень редких случаях (например, в правиле проекции для записей $T\text{-PROJ}$ на рис. 11.7) эти правила вовсе не соблюдаются, поскольку их выполнение привело бы к слишком безобразным или нечитаемым результатам.

Литература

Abadi, Martín. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999. Summary in *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, 1997; volume 1281 of Springer LNCS.

Abadi, Martín, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 147–160, 1999.

Abadi, Martín and Luca Cardelli. On subtyping and matching. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 145–167, 1995.

Abadi, Martín and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

Abadi, Martín, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1–2):9–58, 6 December 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, 1993.

Abadi, Martín, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991a. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 1990.

Abadi, Martín, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991b. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 1989.

Abadi, Martín, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. Summary in *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

Abadi, Martín, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 396–409, 1996.

- Abadi, Martín and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, LICS '96*, pages 242–252. IEEE Computer Society Press, Los Alamitos, CA, July 1996.
- Abelson, Harold and Gerald Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, New York, 1985. Second edition, 1996. Переведена на русский язык, см. Абелсон и Сассман, 2006.
- Abramsky, Samson, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for pcf. *Information and Computation*, 163(2):409–470, December 2000.
- Aczel, Peter. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, number 90 in Studies in Logic and the Foundations of Mathematics, pages 739–782. North Holland, 1977.
- Aczel, Peter. *Non-Well-Founded Sets*. Stanford Center for the Study of Language and Information, 1988. CSLI Lecture Notes number 14.
- Agesen, Ole, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, October 1997.
- Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986. Переведена на русский язык, см. Ахо, Лам, Сети, and Ульман, 2008.
- Aiken, Alexander and Edward L. Wimmers. Type inclusion constraints and type inference. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, 1993.
- Amadio, Roberto M. and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pp. 104–118; also DEC/Compaq Systems Research Center Research Report number 62, August 1990.
- Appel, Andrew W. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- Appel, Andrew W. and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.
- Arbib, Michael and Ernest Manes. *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Press, 1975.
- Ariola, Zena M., Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 233–246, January 1995.

- Arnold, Ken and James Gosling. *The Java Programming Language*. Addison Wesley, 1996. Переведена на русский язык, см. Арнольд and Гослинг, 1997.
- Arnold, Ken, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, and Jim Waldo. *The Jini specification*. Addison-Wesley, Reading, MA, USA, 1999.
- Asperti, Andrea and Giuseppe Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist*. MIT Press, 1991.
- Aspinall, David. Subtyping with singleton types. In *Computer Science Logic (CSL), Kazimierz, Poland*, pages 1–15. Springer-Verlag, 1994.
- Aspinall, David and Adriana Compagnoni. Subtyping dependent types. *Information and Computation*, 266(1–2):273–309, September 2001. Preliminary version in *IEEE Symposium on Logic in Computer Science (LICS)*, 1996.
- Astesiano, Egidio. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer-Verlag, 1991.
- Augustsson, Lennart. A compiler for Lazy ML. In *ACM Symposium on Lisp and Functional Programming (LFP), Austin, Texas*, pages 218–227, August 1984.
- Augustsson, Lennart. Cayenne — a language with dependent types. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 239–250, 1998.
- Baader, Franz and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- Baader, Franz and Jörg Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, Deduction Methodologies, pages 41–125. Oxford University Press, Oxford, UK, 1994.
- Backus, John. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. Reproduced in *Selected Reprints on Dataflow and Reduction Architectures*, ed. S. S. Thakkar, IEEE, 1987, pp. 215–243, and in *ACM Turing Award Lectures: The First Twenty Years*, ACM Press, 1987, pp. 63–130.
- Backus, John. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.
- Bainbridge, E. Stewart, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990. Corrigendum in *TCS* 71(3), 431.
- Baldan, Paolo, Giorgio Ghelli, and Alessandra Raffaetà. Basic theory of F-bounded quantification. *Information and Computation*, 153(1):173–237, 1999.

- Barendregt, Henk P. *The Lambda Calculus*. North Holland, revised edition, 1984.
- Barendregt, Henk P. Functional programming and lambda calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 7, pages 321–364. Elsevier / MIT Press, 1990.
- Barendregt, Henk P. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.
- Barendregt, Henk P. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.
- Barras, Bruno, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.
- Barwise, Jon and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena*. Cambridge University Press, 1996.
- Berardi, Stefano. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt’s cube. Technical report, Department of Computer Science, CMU, and Dipartimento Matematica, Universita di Torino, 1988.
- Berger, Ulrich. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.
- Berger, Ulrich and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ -calculus. In Gilles Kahn, editor, *IEEE Symposium on Logic in Computer Science (LICS)*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
- Birtwistle, Graham M., Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur (Lund, Sweden), Bratt Institut fuer neues Lernen (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.
- Bobrow, Daniel G., Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification X3J13 document 88-002R. *SIGPLAN Notices*, 23, 1988.
- Boehm, Hans-J. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345. IEEE, October 1985.

- Boehm, Hans-J. Type inference in the presence of type abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon*, pages 192–206, June 1989.
- Böhm, Corrado and Alessandro Berarducci. Automatic synthesis of typed Λ -programs on term algebras. *Theoretical Computer Science*, 39(2–3):135–154, August 1985.
- Bono, Viviana and Kathleen Fisher. An imperative first-order calculus with object extension. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.
- Bono, Viviana, Amit J. Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, June 1999a.
- Bono, Viviana, Amit J. Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, April 1999b.
- Bracha, Gilad, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.
- Braithwaite, Richard B. *The Foundations of Mathematics: Collected Papers of Frank P. Ramsey*. Routledge and Kegan Paul, London, 1931.
- Brandt, Michael and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, April 1997. Full version in *Fundamenta Informaticae*, Vol. 33, pp. 309–338, 1998.
- Breazu-Tannen, Val, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Bruce, Kim B. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proceedings of Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, March 1991.
- Bruce, Kim B. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, under the title “Safe type checking in a statically typed object-oriented programming language”.

- Bruce, Kim B. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.
- Bruce, Kim B., Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.
- Bruce, Kim B., Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999. Special issue of papers from *Theoretical Aspects of Computer Software (TACS 1997)*. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.
- Bruce, Kim B. and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.
- Bruce, Kim B. and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992.
- Bruce, Kim B., Leaf Petersen, and Adrian Fiech. Subtyping is not a good “match” for object-oriented languages. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127. Springer-Verlag, 1997.
- Buneman, Peter and Benjamin Pierce. Union types for semistructured data. In *Internet Programming Languages*. Springer-Verlag, September 1998. Proceedings of the International Database Programming Languages Workshop. LNCS 1686.
- Burstall, Rod and Butler Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, 1984.
- Burstall, Rod M. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.
- Canning, Peter, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 457–467, 1989a.
- Canning, Peter, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *ACM Symposium*

- on Functional Programming Languages and Computer Architecture (FPCA), pages 273–280, September 1989b.
- Canning, Peter, Walt Hill, and Walter Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.
- Cardelli, Luca. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation*, 76(2/3):138–164, 1988.
- Cardelli, Luca. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.
- Cardelli, Luca. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, April 1987. An earlier version appeared in the *Polymorphism Newsletter*, January, 1985.
- Cardelli, Luca. Structural subtyping and the notion of power type. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 70–79, January 1988a.
- Cardelli, Luca. Typechecking dependent types and subtypes. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Foundations of Logic and Functional Programming, Workshop Proceedings, Trento, Italy, (Dec. 1986)*, volume 306 of *Lecture Notes in Computer Science*, pages 45–57. Springer-Verlag, 1988b.
- Cardelli, Luca. Notes about $F_{\leq\cdot}^{\omega}$. Unpublished manuscript, October 1990.
- Cardelli, Luca. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC/Compaq Systems Research Center Research Report #45, February 1989.
- Cardelli, Luca. Extensible records in a pure calculus of subtyping. Research report 81, DEC/Compaq Systems Research Center, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Cardelli, Luca. An implementation of $F_{\leq\cdot}$. Research report 97, DEC/Compaq Systems Research Center, February 1993.
- Cardelli, Luca. Type systems. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1996.
- Cardelli, Luca, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research report 52, DEC/Compaq Systems Research Center, November 1989.

- Cardelli, Luca and Xavier Leroy. Abstract types and the dot notation. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods*. North Holland, 1990. Also appeared as DEC/Compaq SRC technical report 56.
- Cardelli, Luca and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Summary in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC/Compaq SRC Research Report 55, Feb. 1990.
- Cardelli, Luca, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1–2): 4–56, 1994. Summary in TACS '91 (Sendai, Japan, pp. 750–770).
- Cardelli, Luca and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC/Compaq Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.
- Cardelli, Luca and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
- Cardone, Felice. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 164–178, Stresa, Italy, July 1989. Springer-Verlag.
- Cardone, Felice and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, 1991.
- Cartwright, Robert and Guy L. Steele, Jr. Compatible genericity with runtime types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Vancouver, British Columbia, SIGPLAN Notices 33(10), pages 201–215. ACM, October 1998.
- Castagna, Giuseppe. *Object-Oriented Programming: A Unified Foundation*. Springer-Verlag, 1997.
- Castagna, Giuseppe, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1): 115–135, 15 February 1995. preliminary version in LISP and Functional Programming, July 1992 (pp. 182–192), and as Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris.
- Chambers, Craig. Object-oriented multi-methods in Cecil. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 33–56, 1992.
- Chambers, Craig. The Cecil language: Specification and rationale. Technical report, University of Washington, March 1993.

- Chambers, Craig and Gary Leavens. Type-checking and modules for multi-methods. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1994. SIGPLAN Notices 29(10).
- Chen, Gang and Giuseppe Longo. Subtyping parametric and dependent types. In Kamareddine et al., editor, *Type Theory and Term Rewriting*, September 1996. Invited lecture.
- Chirimar, Jawahar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, March 1996.
- Church, Alonzo. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936.
- Church, Alonzo. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- Church, Alonzo. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- Clement, Dominique, Joelle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *ACM Conference on LISP and Functional Programming*, pages 13–27, 1986.
- Clinger, William, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.
- Colazzo, Dario and Giorgio Ghelli. Subtyping recursive types in Kernel Fun. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 137–146. IEEE, July 1999.
- Compagnoni, Adriana and Healfdene Goguen. Decidability of higher-order subtyping via logical relations, December 1997a. Manuscript, available at <ftp://www.dcs.ed.ac.uk/pub/hhg/hosdec.ps.gz>.
- Compagnoni, Adriana and Healfdene Goguen. Typed operational semantics for higher order subtyping. Technical Report ECS-LFCS-97-361, University of Edinburgh, July 1997b.
- Compagnoni, Adriana B. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled “Subtyping in F_\wedge^ω is decidable”.
- Compagnoni, Adriana B. and Benjamin C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996. Preliminary version available as University of Edinburgh

- technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993, under the title “Multiple Inheritance via Intersection Types”.
- Constable, Robert L. Types in computer science, philosophy, and logic. In Samuel R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in logic and the foundations of mathematics*, pages 683–786. Elsevier, 1998.
- Constable et al., Robert L. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
- Cook, William. Object-oriented programming versus abstract data types. In J. W. de Bakker et al., editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1991.
- Cook, William and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 433–444, 1989.
- Cook, William R. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
- Cook, William R., Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 125–135, January 1990. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Coppo, Mario and Mariangiola Dezani-Ciancaglini. A new type-assignment for λ -terms. *Archiv Math. Logik*, 19:139–156, 1978.
- Coppo, Mario, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. Functional characterization of some semantic equalities inside λ -calculus. In Hermann A. Maurer, editor, *Proceedings of the 6th Colloquium on Automata, Languages and Programming*, volume 71 of *LNCS*, pages 133–146, Graz, Austria, July 1979. Springer.
- Coquand, Thierry. *Une Théorie des Constructions*. PhD thesis, University Paris VII, January 1985.
- Coquand, Thierry and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
- Courcelle, Bruno. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
- Cousineau, Guy and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
- Crary, Karl. Sound and complete elimination of singleton kinds. Technical Report CMU-CS-00-104, Carnegie Mellon University, School of Computer Science, January 2000.

- Crary, Karl, Robert Harper, and Derek Dreyer. A type system for higher-order modules. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, 2002.
- Crary, Karl, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, May 1999.
- Crary, Karl, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming (ICFP), Baltimore, Maryland, USA*, pages 301–312, 1998.
- Crole, Roy. *Categories for Types*. Cambridge University Press, 1994.
- Curien, Pierre-Louis and Giorgio Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$ -reductions in F_{\leq} . In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 731–749. Springer-Verlag, September 1991.
- Curien, Pierre-Louis and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in F_{\leq} . *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Curry, Haskell B. and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958. Second edition, 1968.
- Damas, Luis and Robin Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 207–212, 1982.
- Danvy, Olivier. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.
- Davey, Brian A. and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
- Davies, Rowan. A refinement-type checker for Standard ML. In *International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
- Davies, Rowan and Frank Pfenning. A modal analysis of staged computation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 258–270, 1996.

- de Bruijn, Nicolas G. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
- de Bruijn, Nicolas G. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.
- Detlefs, David L., K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center (SRC), 1998. Also see <http://research.compaq.com/SRC/esc/overview.html>.
- Donahue, James and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.
- Dowek, Gilles, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.
- Drossopoulou, Sophia, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theory and Practice of Object Systems*, 7(1):3–24, 1999. Summary in European Conference on Object-Oriented Programming (ECOOP), 1997.
- Duggan, Dominic and Adriana Compagnoni. Subtyping for object type constructors. In *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, January 1999.
- Dybvig, R. Kent. *The Scheme Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, second edition, 1996. Available electronically at <http://www.scheme.com/tsp12d/>.
- Eidorff, Peter, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine B. Sørensen, and Mads Tofte. AnnoDomini in practice: A type-theoretic approach to the Year 2000 problem. In Jean-Yves Girard, editor, *Proc. Symposium on Typed Lambda Calculus and Applications (TLCA)*, volume 1581 of *Lecture Notes in Computer Science*, pages 6–13, L’Aquila, Italy, April 1999. Springer-Verlag.
- Eifrig, Jonathan, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.
- Feinberg, Neal, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.
- Felleisen, Matthias and Daniel P. Friedman. *A Little Java, A Few Patterns*. MIT Press, Cambridge, Massachusetts, 1998.

Felty, Amy, Elsa Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. Lambda prolog: An extended logic programming language. In E. Lusk; R. Overbeek, editor, *Proceedings on the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 754–755, Berlin, May 1988. Springer.

Filinski, Andrzej. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as technical report BRICS RS-99-17.

Filinski, Andrzej. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

Fisher, Kathleen. Classes = objects + data abstraction. In Kim Bruce and Giuseppe Longo, editors, *Workshop on Foundations of Object-Oriented Languages (FOOL), informal proceedings*, July 1996a. Invited talk. Also available as Stanford University Technical Note STAN-CS-TN-96-31.

Fisher, Kathleen. *Type Systems for object-oriented programming languages*. PhD thesis, Stanford University, 1996b. STAN-CS-TR-98-1602.

Fisher, Kathleen, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing* (formerly *BIT*), 1:3–37, 1994. Summary in *Proc. IEEE Symp. on Logic in Computer Science*, 1993, 26–38.

Fisher, Kathleen and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

Fisher, Kathleen and John C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, 1998.

Fisher, Kathleen and John H. Reppy. The design of a class mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 37–49, 1999.

Flanagan, Cormac and Matthias Felleisen. Componential set-based analysis. *ACM SIGPLAN Notices*, 32(5):235–248, May 1997.

Flatt, Matthew and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 236–248, 1998.

Flatt, Matthew, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 1998a.

- Flatt, Matthew, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR97-293, Computer Science Department, Rice University, February 1998b. Corrected June, 1999.
- Freeman, Tim and Frank Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario*, June 1991.
- Frege, Gottlob. *Begriffschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle: L. Nebert, 1879. Available in several translations.
- Friedman, Daniel P. and Matthias Felleisen. *The Little Schemer*. MIT Press, 1996.
- Friedman, Daniel P., Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. McGraw-Hill Book Co., New York, N.Y., second edition, 2001.
- Friedman, Harvey. Equality between functionals. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 22–37, Berlin, 1975. Springer-Verlag.
- Gallier, Jean. On Girard's "Candidats de reductibilité". In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing, pages 123–203. Academic Press, 1990.
- Gallier, Jean. Constructive logics. Part I: A tutorial on proof systems and typed λ -calculi. *Theoretical Computer Science*, 110(2):249–339, March 1993.
- Gandy, Robin O. The simple theory of types. In *Logic Colloquium 76*, volume 87 of *Studies in Logic and the Foundations of Mathematics*, pages 173–181. North Holland, 1976.
- Gapeyev, Vladimir, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. In *International Conference on Functional Programming (ICFP), Montreal, Canada*, 2000. To appear in *Journal of Functional Programming*.
- Garrigue, Jaques and Hassan Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 35–47, 1994.
- Garrigue, Jaques and Didier Rémy. Extending ML with semi-explicit polymorphism. In Martín Abadi and Takayasu Ito, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, pages 20–46. Springer-Verlag, September 1997.
- Ghelli, Giorgio. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, March 1990. Technical report TD-6/90, Dipartimento di Informatica, Università di Pisa.

- Ghelli, Giorgio. Recursive types are not conservative over F_{\leq} . In M. Bezen and J.F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in Lecture Notes in Computer Science, pages 146–162, Berlin, March 1993. Springer-Verlag.
- Ghelli, Giorgio. Divergence of F_{\leq} type checking. *Theoretical Computer Science*, 139(1,2):131–162, 1995.
- Ghelli, Giorgio. Termination of system F-bounded: A complete proof. *Information and Computation*, 139(1):39–56, 1997.
- Ghelli, Giorgio and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998.
- Gifford, David, Pierre Jouvelot, John Lucassen, and Mark Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.
- Girard, Jean-Yves. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris VII, 1972. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).
- Girard, Jean-Yves. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- Girard, Jean-Yves, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
- Glew, Neal. Type dispatch for named hierarchical types. In *International Conference on Functional Programming (ICFP), Paris, France*, pages 172–182, 1999.
- Gordon, Andrew. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, pages 78–95. Springer Workshops in Computing, 1995.
- Gordon, Michael J. Adding eval to ML. Manuscript, circa 1980.
- Gordon, Michael J., Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.
- Goto, Eiichi. Monocopy and associative algorithms in extended Lisp. Technical Report TR 74-03, University of Tokyo, May 1974.
- Goubault-Larrecq, Jean and Ian Mackie. *Proof Theory and Automated Deduction (Applied Logic Series, V. 6)*. Kluwer, 1997.
- Grattan-Guinness, Ivor. *The search for mathematical roots, 1870–1940: Logics, set theories and the foundations of mathematics from Cantor through Russell to Gödel*. Princeton University Press, 2001.
- Gries, David, editor. *Programming Methodology*. Springer-Verlag, New York, 1978.

- Gunter, Carl A. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
- Gunter, Carl A. and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.
- Hall, Cordelia V., Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.
- Halmos, Paul R. *Naive Set Theory*. Springer, New York, 1987.
- Harper, Robert. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also (Harper, 1996).
- Harper, Robert. A note on: “A simplified account of polymorphic references” [Inform. Process. Lett. 51 (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, January 1996. See (Harper, 1994).
- Harper, Robert, Bruce Duba, and David MacQueen. First-class continuations in ML. *Journal of Functional Programming*, 3(4), October 1993. Short version in POPL ’91.
- Harper, Robert, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Summary in LICS’87.
- Harper, Robert and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, January 1994.
- Harper, Robert, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 341–354, January 1990.
- Harper, Robert and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 130–141, 1995.
- Harper, Robert and Benjamin Pierce. A record calculus based on symmetric concatenation. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 131–142, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.
- Harper, Robert and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
- Hasegawa, Ryu. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In Takayasu Ito and Albert Meyer, editors, *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, 1991.

- Hayashi, Susumu. Singleton, union and intersection types for program extraction.
In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science, pages 701–730. Springer-Verlag, September 1991. Full version in *Information and Computation*, 109(1/2):174–210, 1994.
- Henglein, Fritz. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
- Henglein, Fritz. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994. Selected papers of the Fourth European Symposium on Programming (Rennes, 1992).
- Henglein, Fritz and Harry G. Mairson. The complexity of type inference for higher-order typed lambda-calculi. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 119–130, January 1991.
- Hennessy, Matthew. *A Semantics of Programming Languages: An Elementary Introduction Using Operational Semantics*. John Wiley and Sons, 1990. Currently out of print; available from <http://www.cogs.susx.ac.uk/users/matthewh/seminotes.ps.gz>.
- Hennessy, Matthew and James Riely. Resource access control in systems of mobile agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier Science Publishers, 1998. Full version available as CogSci Report 2/98, University of Sussex, Brighton.
- Hindley, J. Roger. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
- Hindley, J. Roger. Types with intersection, an introduction. *Formal Aspects of Computing*, 4:470–486, 1992.
- Hindley, J. Roger. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1997.
- Hindley, J. Roger and Jonathan P. Seldin. *Introduction to Combinators and λ -Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.
- Hoang, My, John Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 15–25. IEEE Computer Society Press, 1993.
- Hodas, J. S. Lolli: An extension of λ Prolog with linear context management.
In D. Miller, editor, *Workshop on the λ Prolog Programming Language*, pages 159–168, Philadelphia, Pennsylvania, August 1992.
- Hofmann, Martin. Syntax and semantics of dependent types. In *Semantics and Logic of Computation*. Cambridge University Press, 1997.

Hofmann, Martin and Benjamin Pierce. Positive subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 186–197, January 1995a. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

Hofmann, Martin and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995b. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title “An Abstract View of Objects and Subtyping (Preliminary Report),” as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

Hofmann, Martin and Benjamin C. Pierce. Type destructors. In Didier Rémy, editor, *Informal proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998. Full version to appear in *Information and Computation*.

Hook, J.G. Understanding Russell – a first attempt. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), Springer LNCS 173*, pages 69–85. Springer-Verlag, 1984.

Hopcroft, John E. and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

Hosoya, Haruo and Benjamin Pierce. Regular expression pattern matching. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.

Hosoya, Haruo and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

Hosoya, Haruo and Benjamin C. Pierce. XDUCE: A typed XML processing language (preliminary report). In *International Workshop on the Web and Databases (WebDB)*, May 2000.

Hosoya, Haruo, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2001. To appear; short version in ICFP 2000.

Howard, William A. Hereditarily majorizable functionals of finite type. In Anne Sjerp Troelstra, editor, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, pages 454–461. Springer-Verlag, Berlin, 1973. Appendix.

Howard, William A. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

Howe, Douglas. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.

- Hudak, Paul, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.
- Huet, Gérard. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
- Huet, Gérard. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω* . Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.
- Huet, Gérard, editor. *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.
- Hyland, J. Martin E. and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, December 2000.
- Igarashi, Atsushi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version to appear in ACM Transactions on Programming Languages and Systems (TOPLAS), 2001.
- Igarashi, Atsushi and Benjamin C. Pierce. On inner classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2000. Also in informal proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL). To appear in *Information and Computation*.
- Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.
- Ishtiaq, Samin and Peter O'Hearn. Bi as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages (POPL)*, London, England, 2001.
- Jacobs, Bart. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier, 1999.
- Jagannathan, Suresh and Andrew Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *LNCS*, pages 207–224. Springer-Verlag, 1995.
- Jay, C. Barry and Milan Sekanina. Shape checking of array programs. In *Computing: The Australasian Theory Seminar (Proceedings)*, volume 19 of *Australian Computer Science Communications*, pages 113–121, 1997.
- Jim, Trevor. Rank-2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1995.
- Jim, Trevor. What are principal typings and what are they good for? In ACM, editor, *ACM Symposium on Principles of Programming Languages (POPL)*, St. Petersburg Beach, Florida, pages 42–53, 1996.

- Jim, Trevor and Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.
- Jones, Mark P. ML typing, explicit polymorphism, and qualified types, 1994a.
- Jones, Mark P. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994b.
- Jones, Richard and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.
- Jouvelot, Pierre and David Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 303–310, January 1991.
- Butting, L.S. van Benthem, James McKinna, and Robert Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, pages 19–61, Nijmegen, The Netherlands, May 1994. Springer-Verlag LNCS 806.
- Kaes, Stefan. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1988.
- Kahn, Gilles. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
- Kamin, Samuel N. Inheritance in Smalltalk-80: A denotational definition. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 80–87, January 1988.
- Kamin, Samuel N. and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.
- Katiyar, Dinesh, David Luckham, and John Mitchell. A type system for prototyping languages. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 138–150, January 1994.
- Katiyar, Dinesh and Sriram Sankar. Completely bounded quantification is decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.
- Kelsey, Richard, William Clinger, and Jonathan Rees. Revised⁵ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1): 7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

- Kennedy, Andrew. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems—ESOP’94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362, Edinburgh, U.K., 11–13 April 1994. Springer.
- Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, second edition, 1988. Переведена на русский язык, см. Керніган and Рітчі, 2009.
- Kfoury, Assaf J., Harry Mairson, Franklyn Turbak, and Joe B. Wells. Relating typability and expressiveness in finite-rank intersection type systems. In *International Conference on Functional Programming (ICFP), Paris, France*, volume 34.9 of *ACM Sigplan Notices*, pages 90–101, N.Y., September 27–29 1999. ACM Press.
- Kfoury, Assaf J. and Jerzy Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ -calculus. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 2–11, Philadelphia, PA, June 1990. Full version in *Information and Computation*, 98(2), 228–257, 1992.
- Kfoury, Assaf J., Jerzy Tiuryn, and Paweł Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Colloq. on Trees in Algebra and Programming*, pages 206–220. Springer LNCS 431, 1990.
- Kfoury, Assaf J., Jerzy Tiuryn, and Paweł Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993a.
- Kfoury, Assaf J., Jerzy Tiuryn, and Paweł Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993b. Summary in *STOC 1990*.
- Kfoury, Assaf J., Jerzy Tiuryn, and Paweł Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.
- Kfoury, Assaf J. and Joe B. Wells. Principality and decidable type inference for finite-rank intersection types. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 161–174, New York, NY, January 1999. ACM.
- Kiczales, Gregor, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
- Kirchner, Claude and Jean-Pierre Jouannaud. Solving equations in abstract algebras: a rule-based survey of unification. Research Report 561, Université de Paris Sud, Orsay, France, April 1990.
- Klop, Jan W. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

- Kobayashi, Naoki, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida, 1996*. Full version in *ACM Transactions on Programming Languages and Systems*, 21(5), pp. 914–947, September 1999.
- Kozen, Dexter, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, pages 419–428, 1993.
- Laan, Twan Dismas Laurens. *The Evolution of Type Theory in Logic and Mathematics*. PhD thesis, Techn. Univ. Eindhoven, 1997.
- Landin, Peter J. The mechanical evaluation of expressions. *Computer Journal*, 6: 308–320, January 1964.
- Landin, Peter J. A correspondence between ALGOL 60 and Church's lambda-notation: Parts I and II. *Communications of the ACM*, 8(2,3):89–101, 158–165, February and March 1965.
- Landin, Peter J. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.
- Lassez, Jean-Louis and Gordin Plotkin, editors. *Computational Logic, Essays in Honor of Alan Robinson*. MIT Press, 1991.
- Läufer, Konstantin. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, 1992.
- Läufer, Konstantin and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1411–1430, September 1994. Summary in *Phoenix Seminar and Workshop on Declarative Programming*, Nov. 1991.
- League, Christopher, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *International Conference on Functional Programming (ICFP), Paris, France*, September 1999.
- League, Christopher, Valery Trifonov, and Zhong Shao. Type-preserving compilation of Featherweight Java. In *Foundations of Object-Oriented Languages (FOOL8)*, London, January 2001.
- Lee, Oukseh and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.
- Leivant, Daniel. Polymorphic type inference. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*. ACM, 1983.
- Lemmon, E. John, Carew A. Meredith, David Meredith, Arthur N. Prior, and Ivo Thomas. Calculi of pure strict implication, 1957. Mimeographed version, 1957; published in *Philosophical Logic*, ed. Davis, Hockney, and Wilson, D. Reidel Co., Netherlands, 1969, pp. 215–250.

- Leroy, Xavier. Manifest types, modules and separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 109–122, Portland, OR, January 1994.
- Leroy, Xavier. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from <http://caml.inria.fr>.
- Leroy, Xavier and Michel Mauny. Dynamics in ML. In John Hughes, editor, *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA) 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 406–426. Springer-Verlag, 1991.
- Leroy, Xavier and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, March 2000. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, 1999.
- Leroy, Xavier and François Rouaix. Security properties of typed applets. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 391–403, January 1998.
- Leroy, Xavier and Pierre Weis. Polymorphic type inference and assignment. In *ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 291–302, 1991.
- Lescanne, Pierre and Jocelyn Rouyer-Degli. Explicit substitutions with de Bruijn's levels. In J. Hsiang, editor, *Proceedings of the 6th Conference on Rewriting Techniques and Applications (RTA), Kaiserslautern (Germany)*, volume 914, pages 294–308, 1995.
- Levin, Michael Y. and Benjamin C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 2001. To appear. A preliminary version appeared as an invited talk at the *Logical Frameworks and Metalanguages Workshop (LFM)*, June 2000.
- Lillibridge, Mark. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997.
- Liskov, Barbara, Russell Atkinson, Toby Bloom, Elliott Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.
- Liskov, Barbara, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems*.
- Luo, Zhaohui. *Computation and Reasoning: A Type Theory for Computer Science*. Number 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

Luo, Zhaohui and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

Ma, QingMing. Parametricity as subtyping. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992.

Mackie, Ian. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, October 1994.

MacQueen, David. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 277–286, January 1986.

MacQueen, David, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

MacQueen, David B. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, 1986.

Magnusson, Lena and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

Mairson, Harry G. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 382–401. ACM Press, New York, 1990.

Martin-Löf, Per. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118. North-Holland, Amsterdam, 1973.

Martin-Löf, Per. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI*. North Holland, Amsterdam, 1982.

Martin-Löf, Per. *Intuitionistic Type Theory*. Bibliopolis, 1984.

Martini, Simone. Bounded quantifiers have interval models. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. ACM.

McCarthy, John. History of LISP. In R. L. Wexelblatt, editor, *History of Programming Languages*, pages 173–197. Academic Press, New York, 1981.

McCarthy, John, S. R. Russell, D. Edwards, et al. *LISP Programmer's Manual*. Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, November 1959. Handwritten Draft + Machine Typed.

- McKinna, James and Robert Pollack. Pure Type Systems formalized. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag LNCS 664, March 1993.
- Meertens, Lambert. Incremental polymorphic type checking in B. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 1983.
- Milner, Robin. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.
- Milner, Robin. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- Milner, Robin. *Communication and Concurrency*. Prentice Hall, 1989.
- Milner, Robin. The polyadic π -calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.
- Milner, Robin. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
- Milner, Robin, Joachim Parrow, and David Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.
- Milner, Robin and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991a.
- Milner, Robin and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1991b.
- Milner, Robin, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
- Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- Mitchell, John C. Coercion and type inference (summary). In *ACM Symposium on Principles of Programming Languages (POPL), Salt Lake City, Utah*, pages 175–185, January 1984a.
- Mitchell, John C. Type inference and type containment. In *Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)*, pages 257–278, Berlin, June 1984b. Springer LNCS 173. Full version in *Information and Computation*, vol. 76, no. 2/3, 1988, pp. 211–249. Reprinted in *Logical Foundations of Functional Programming*, ed. G. Huet, Addison-Wesley (1990) 153–194.

- Mitchell, John C. Representation independence and data abstraction (preliminary version). In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 263–276, 1986.
- Mitchell, John C. Toward a typed foundation for method specialization and inheritance. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 109–124, January 1990a. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Mitchell, John C. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. North-Holland, Amsterdam, 1990b.
- Mitchell, John C. *Foundations for Programming Languages*. MIT Press, Cambridge, Massachusetts, 1996. Переведена на русский язык, см. [Митчелл, 2010](#).
- Mitchell, John C. and Robert Harper. The essence of ML. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 1988. Full version in *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 2, 1993, pp. 211–252, under the title “On the type structure of Standard ML”.
- Mitchell, John C. and Albert R. Meyer. Second-order logical relations (extended abstract). In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 225–236, Berlin, 1985. Springer-Verlag.
- Mitchell, John C. and Gordon D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Summary in *ACM Symposium on Principles of Programming Languages (POPL), New Orleans, Louisiana*, 1985.
- Morris, James H. Lambda calculus models of programming languages. Technical Report MIT-LCS//MIT/LCS/TR-57, Massachusetts Institute of Technology, Laboratory for Computer Science, December 1968.
- Morrisett, Greg, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 66–77, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.
- Morrisett, Greg, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 85–97, January 1998.
- Mugridge, Warwick B., John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In Pierre America, editor, *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 307–324. Springer-Verlag, 1991.

- Mycroft, Alan. Dynamic types in ML. Manuscript, 1983.
- Mycroft, Alan. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228, Toulouse, France, April 1984. Springer.
- Myers, Andrew C., Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 132–145, January 1997.
- Nadathur, Gopalan and Dale Miller. An overview of λ Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, MIT Press, Cambridge, Massachusetts, August 1988.
- Naur, Peter et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6:1–17, January 1963.
- Necula, George C. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 106–119, 15–17 January 1997.
- Necula, George C. and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996, Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX press.
- Necula, George C. and Peter Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1998.
- Nelson, Greg, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.
- Nipkow, Tobias and David von Oheimb. Java_{light} is type-safe — definitely. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 161–170, January 1998.
- O'Callahan, Robert and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 338–348. ACM Press, 1997.
- Odersky, Martin. Functional nets. In *Proc. European Symposium on Programming (ESOP)*, pages 1–25. Springer-Verlag, 2000. Lecture Notes in Computer Science 1782.
- Odersky, Martin and Konstantin Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 54–67, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

- Odersky, Martin, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. Summary in *Workshop on Foundations of Object-Oriented Languages (FOOL)*, *informal proceedings*, 1997.
- Odersky, Martin and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 146–159, January 1997.
- Odersky, Martin and Christoph Zenger. Nested types. In *Workshop on Foundations of Object-Oriented Languages (FOOL 8)*, January 2001.
- Odersky, Martin, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, March 2001.
- O’Hearn, Peter W., Makoto Takeyama, A. John Power, and Robert D. Tennent. Syntactic control of interference revisited. In *MFPS XI, conference on Mathematical Foundations of Program Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, March 1995.
- O’Toole, James W. and David K. Gifford. Type reconstruction with first-class polymorphic values. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon*, pages 207–217, June 1989.
- Palsberg, Jens and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 197–208, 1998.
- Palsberg, Jens and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.
- Park, David. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, Berlin, 1981.
- Paulin-Mohring, Christine. Extracting F_ω ’s programs from proofs in the calculus of constructions. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 89–104, January 1989.
- Paulson, Laurence C. *ML for the Working Programmer*. Cambridge University Press, New York, NY, second edition, 1996.
- Perry, Nigel. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.
- Peyton Jones, Simon L. and David R. Lester. *Implementing Functional Languages*. Prentice Hall, 1992.

- Pfenning, Frank. Partial polymorphic type inference and higher-order unification. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 153–163, July 1988. Also available as Ergo Report 88–048, School of Computer Science, Carnegie Mellon University, Pittsburgh.
- Pfenning, Frank. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.
- Pfenning, Frank. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993a. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.
- Pfenning, Frank. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993b.
- Pfenning, Frank. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814.
- Pfenning, Frank. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.
- Pfenning, Frank. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 1999.
- Pfenning, Frank. *Computation and Deduction*. Cambridge University Press, 2001.
- Pfenning, Frank and Peter Lee. Metacircularity in the polymorphic λ -calculus. *Theoretical Computer Science*, 89(1):137–159, 21 October 1991. Summary in *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359, Springer-Verlag LNCS 352, March 1989.
- Pierce, Benjamin C. *Basic Category Theory for Computer Scientists*. MIT Press, 1991a.
- Pierce, Benjamin C. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991b. Available as School of Computer Science technical report CMU-CS-91-205.
- Pierce, Benjamin C. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). Summary in *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*.

Pierce, Benjamin C. Even simpler type-theoretic foundations for OOP. Manuscript (circulated electronically), March 1996.

Pierce, Benjamin C. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997a.

Pierce, Benjamin C. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997b. Summary in *Typed Lambda Calculi and Applications*, March 1993, pp. 346–360.

Pierce, Benjamin C. and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.

Pierce, Benjamin C. and Martin Steffens. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

Pierce, Benjamin C. and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

Pierce, Benjamin C. and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, 1993.

Pierce, Benjamin C. and David N. Turner. Local type argument synthesis with bounded quantification. Technical Report 495, Computer Science Department, Indiana University, January 1997.

Pierce, Benjamin C. and David N. Turner. Local type inference. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, 1998. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

Pierce, Benjamin C. and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

Pitts, Andrew M. Polymorphism is set theoretic, constructively. In Pitt, Poigné, and Rydeheard, editors, *Category Theory and Computer Science, Edinburgh*, pages 12–39. Springer-Verlag, 1987. LNCS volume 283.

Pitts, Andrew M. Non-trivial power types can't be subtypes of polymorphic types. In *Fourth Annual Symposium on Logic in Computer Science, Pacific Grove, California*, pages 6–13. IEEE, June 1989.

- Pitts, Andrew M. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.
- Plasmeijer, Marinus J. CLEAN: a programming environment based on term graph rewriting. *Theoretical Computer Science*, 194(1–2), March 1998.
- Plotkin, Gordon. Call-by-name, call-by-value, and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- Plotkin, Gordon and Martín Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in Lecture Notes in Computer Science, pages 361–375. Springer-Verlag, March 1993.
- Plotkin, Gordon, Martín Abadi, and Luca Cardelli. Subtyping and parametricity. In *Proceedings of the Ninth IEEE Symposium on Logic in Computer Science*, pages 310–319, 1994.
- Plotkin, Gordon D. Lambda-definability and logical relations. Memorandum SAI-RM-4, University of Edinburgh, Edinburgh, Scotland, October 1973.
- Plotkin, Gordon D. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
- Plotkin, Gordon D. Lambda-definability in the full type hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, London, 1980.
- Plotkin, Gordon D. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.
- Poll, Erik. Width-subtyping and polymorphic record update. Manuscript, June 1996.
- Pollack, Robert. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.
- Pollack, Robert. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
- Pottier, François. Simplifying subtyping constraints. In *International Conference on Functional Programming (ICFP), Amsterdam, The Netherlands*, 1997.
- Pottinger, Garrell. A type assignment for the strongly normalizable λ -terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. Academic Press, New York, 1980.
- Quine, Willard V. *Quiddities: An Intermittently Philosophical Dictionary*. Harvard University Press, Cambridge, MA, 1987.

- Ramsey, Frank P. The foundations of mathematics. *Proceedings of the London Mathematical Society, Series 2*, 25(5):338–384, 1925. Reprinted in (Braithwaite, 1931).
- Ranta, Aarne. *Type-Theoretical Grammar*. Clarendon Press, Oxford, 1995.
- Reade, Chris. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, Wokingham, England, 1989.
- Reddy, Uday S. Objects as closures: Abstract semantics of object oriented languages. In *ACM Symposium on Lisp and Functional Programming (LFP), Snowbird, Utah*, pages 289–297, Snowbird, Utah, July 1988.
- Relax. Document Description and Processing Languages — Regular Language Description for XML (RELAX) — Part 1: RELAX Core. Technical Report DTR 22250-1, ISO/IEC, October 2000.
- Rémy, Didier. Typechecking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 242–249, January 1989. Long version in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Rémy, Didier. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels*. PhD thesis, Université Paris VII, 1990.
- Rémy, Didier. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992a.
- Rémy, Didier. Projective ML. In *ACM Symposium on Lisp and Functional Programming (LFP)*, pages 66–75, 1992b.
- Rémy, Didier. Typing record concatenation for free. In *ACM Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, January 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Rémy, Didier. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.
- Rémy, Didier. *Des enregistrements aux objets*. Mémoire d’habilitation à diriger des recherches, Université de Paris 7, 1998. In English, except for introductory chapter; includes (Rémy, 1989) and (Rémy, 1992b).
- Rémy, Didier and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. Summary in *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, 1997.

- Reynolds, John. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.
- Reynolds, John C. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.
- Reynolds, John C. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages 1975*, pages 157–168, Rocquencourt, France, 1975. IFIP Working Group 2.1 on Algol, INRIA. Reprinted in (Gries, 1978, pages 309–317) and (Gunter and Mitchell, 1994, pages 13–23).
- Reynolds, John C. Syntactic control of interference. In *ACM Symposium on Principles of Programming Languages (POPL), Tucson, Arizona*, pages 39–46, 1978. Reprinted in O’Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 273–286, Birkhäuser, 1997.
- Reynolds, John C. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science. Springer-Verlag, January 1980. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Reynolds, John C. *The Craft of Programming*. Prentice-Hall International, London, 1981.
- Reynolds, John C. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).
- Reynolds, John C. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156, Berlin, 1984. Springer-Verlag.
- Reynolds, John C. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988. Reprinted in O’Hearn and Tennent, *ALGOL-like Languages*, vol. 1, pages 173–233, Birkhäuser, 1997.
- Reynolds, John C. Syntactic control of interference, part 2. Report CMU-CS-89-130, Carnegie Mellon University, April 1989.
- Reynolds, John C. Introduction to part II, polymorphic lambda calculus. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, Reading, Massachusetts, 1990.
- Reynolds, John C. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai,*

- Japan)*, number 526 in Lecture Notes in Computer Science, pages 675–700. Springer-Verlag, September 1991.
- Reynolds, John C. Normalization and functor categories. In Olivier Danvy and Peter Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, (Chalmers, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998a.
- Reynolds, John C. *Theories of Programming Languages*. Cambridge University Press, 1998b.
- Reynolds, John C. and Gordon Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105(1):1–29, 1993. Summary in (Huet, 1990).
- Robinson, Edmund and Robert Tennent. Bounded quantification and record-update problems. Message to Types electronic mail list, October 1988.
- Robinson, J. Alan. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.
- Russell, Bertrand. Letter to Frege, 1902. Reprinted (in English) in J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*; Harvard University Press, Cambridge, MA, 1967; pages 124–125.
- Schaffert, Justin Craig. A formal definition of CLU. Master's thesis, MIT, January 1978. MIT/LCS/TR-193.
- Scheifler, Robert William. A denotational semantics of CLU. Master's thesis, MIT, May 1978. MIT/LCS/TR-201.
- Schmidt, David A. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.
- Schmidt, David A. *The Structure of Typed Programming Languages*. MIT Press, 1994.
- Schönfinkel, Moses. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924. Translated into English and republished as “On the building blocks of mathematical logic” in (van Heijenoort, 1967, pp. 355–366).
- Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 1999.
- Severi, Paula and Erik Poll. Pure type systems with definitions. In *Proceedings of Logical Foundations of Computer Science (LFCS)*, pages 316–328. Springer-Verlag, 1994. LNCS volume 813.
- Shalit, Andrew. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.
- Shields, Mark. *Static Types for Dynamic Documents*. PhD thesis, Department of Computer Science, Oregon Graduate Institute, February 2001.

- Simmons, Harold. *Derivation and Computation : Taking the Curry-Howard Correspondence Seriously*. Number 51 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2000.
- Smith, Frederick, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer-Verlag, April 2000.
- Smith, Jan, Bengt Nordström, and Kent Petersson. *Programming in Martin-Löf’s Type Theory. An Introduction*. Oxford University Press, 1990.
- Solomon, Marvin. Type definitions with parameters. In *ACM Symposium on Principles of Programming Languages (POPL), Tucson, Arizona*, pages 31–38, January 23–25, 1978.
- Sommaruga, Giovanni. *History and Philosophy of Constructive Type Theory*, volume 290 of *Synthese Library*. Kluwer Academic Pub., 2000.
- Somogyi, Zoltan, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–November 1996.
- Sørensen, Morten Heine and Paweł Urzyczyn. Lectures on the Curry-Howard isomorphism. Technical Report 98/14 (= TOPPS note D-368), DIKU, Copenhagen, 1998.
- Statman, Richard. Completeness, invariance and λ -definability. *Journal of Symbolic Logic*, 47(1):17–26, 1982.
- Statman, Richard. Equality between functionals, revisited. In *Harvey Friedman’s Research on the Foundations of Mathematics*, pages 331–338. North-Holland, Amsterdam, 1985a.
- Statman, Richard. Logical relations and the typed λ -calculus. *Information and Control*, 65(2–3):85–97, May–June 1985b.
- Steffen, Martin. *Polarized Higher-Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1998.
- Stone, Christopher A. and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 214–227, January 19–21, 2000.
- Strachey, Christopher. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.
- Stroustrup, Bjarne. *The C++ Programming Language*. Addison Wesley Longman, Reading, MA, third edition, 1997. Переведена на русский язык, см. Страпу́струп, 2008.

- Studer, Thomas. Constructive foundations for featherweight java. In R. Kahle, P. Schroeder-Heister, and R. Stärk, editors, *Proof Theory in Computer Science*. Springer-Verlag, 2001. Lecture Notes in Computer Science, volume 2183.
- Sumii, Eijiro and Benjamin C. Pierce. Logical relations for encryption. In *Computer Security Foundations Workshop*, June 2001. Submitted (by invitation) to *Journal of Computer Security*.
- Sussman, Gerald Jay and Guy Lewis Steele, Jr. Scheme: an interpreter for extended lambda calculus. MIT AI Memo 349, Massachusetts Institute of Technology, December 1975. Reprinted, with a foreword, in *Higher-Order and Symbolic Computation*, 11(4), pp. 405–439, 1998.
- Syme, Don. Proving Java type soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.
- Tait, William W. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.
- Tait, William W. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251, Boston, 1975. Springer-Verlag.
- Talpin, Jean-Pierre and Pierre Jouvelot. The type and effects discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 162–173, 1992.
- Tarditi, David, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL : A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania*, pages 181–192, May 21–24 1996.
- Tarski, Alfred. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- Tennent, Robert D. *Principles of Programming Languages*. Prentice-Hall, 1981.
- Terlouw, J. Een nadere bewijstheoretische analyse van GSTTs. Manuscript, University of Nijmegen, Netherlands, 1989.
- Thatte, Satish R. Quasi-static typing (preliminary report). In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 367–381, 1990.
- Thompson, Simon. *Type Theory and Functional Programming*. Addison Wesley, 1991.
- Thompson, Simon. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999.
- Tiuryn, Jerzy. Type inference problems: A survey. In B. Rovan, editor, *Mathematical Foundations of Computer Science 1990, Banská Bystrica, Czechoslovakia*, volume 452 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, New York, NY, 1990.

- Tofte and Birkedal. A region inference algorithm. *ACM TOPLAS: ACM Transactions on Programming Languages and Systems*, 20, 1998.
- Tofte, Mads. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.
- Tofte, Mads and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, January 1994.
- Tofte, Mads and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 February 1997.
- Trifonov, Valery and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. Springer-Verlag, September 1996.
- Turner, David N., Philip Wadler, and Christian Mossin. Once upon a type. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, San Diego, California, 1995.
- Turner, Raymond. *Constructive Foundations for Functional Languages*. McGraw Hill, 1991.
- Ullman, Jeffrey D. *Elements of ML Programming*. Prentice-Hall, ML97 edition, 1997.
- Ungar, David and Randall B. Smith. Self: The power of simplicity. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 227–241, 1987.
- U.S. Dept. of Defense. *Reference Manual for the Ada Programming Language*. GPO 008-000-00354-8, 1980.
- van Benthem, Johan. *Language in Action: Categories, Lambdas, and Dynamic Logic*. MIT Press, 1995.
- van Benthem, Johan F. A. K. and Alice Ter Meulen, editors. *Handbook of Logic and Language*. MIT Press, 1997.
- van Heijenoort, Jan, editor. *From Frege to Gödel*. Harvard University Press, Cambridge, Massachusetts, 1967.
- van Wijngaarden, Adriaan, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5(1–3):1–236, 1975.
- Vouillon, Jérôme. *Conception et réalisation d'une extension du langage ML avec des objets*. PhD thesis, Université Paris 7, October 2000.

- Vouillon, Jérôme. Combining subsumption and binary methods: An object calculus with views. In *ACM Symposium on Principles of Programming Languages (POPL), London, England*, 2001.
- Wadler, Philip. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989. Imperial College, London.
- Wadler, Philip. Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, pages 546–566, 1990.
- Wadler, Philip. Is there a use for linear logic? In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273, 1991.
- Wadler, Philip. New languages, old logic. *Dr. Dobbs Journal*, December 2000.
- Wadler, Philip. The Girard-Reynolds isomorphism. In Naoki Kobayashi and Benjamin Pierce, editors, *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, Lecture Notes in Computer Science. Springer-Verlag, 2001.
- Wadler, Philip and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 60–76, 1989.
- Wadsworth, Christopher P. *Semantics and pragmatics of the lambda-calculus*. PhD thesis, Programming Research Group, Oxford University, 1971.
- Wand, Mitchell. Finding the source of type errors. *13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 38–43, 1986.
- Wand, Mitchell. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.
- Wand, Mitchell. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.
- Wand, Mitchell. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89-2, College of Computer Science, Northeastern University, February 1989a. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).
- Wand, Mitchell. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989b.
- Weis, Pierre, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.

- Wells, Joe B. Typability and type checking in the second-order λ -calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 176–185, 1994.
- Whitehead, Alfred North and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, 1910. Three volumes (1910; 1912; 1913).
- Wickline, Philip, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), September 1998. Article 8.
- Wille, Christoph. *Presenting C#*. SAMS Publishing, 2000.
- Winskel, Glynn. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.
- Wirth, Niklaus. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971.
- Wright, Andrew K. Typing references by effect inference. In Bernd Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer-Verlag, New York, N.Y., 1992.
- Wright, Andrew K. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.
- Wright, Andrew K. and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.
- Xi, Hongwei and Robert Harper. A dependently typed assembly language. In *International Conference on Functional Programming (ICFP), Firenze, Italy*, 2001.
- Xi, Hongwei and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, pages 249–257, 1998.
- Xi, Hongwei and Frank Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, ACM SIGPLAN Notices, pages 214–227, 1999.
- XML 1998. Extensible markup language (XMLTM), February 1998. XML 1.0, W3C Recommendation, <http://www.w3.org/XML/>.
- XS 2000. XML Schema Part 0: Primer, W3C Working Draft. <http://www.w3.org/TR/xmlschema-0/>, 2000.
- Yelick, Kathy, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

Zwanenburg, Jan. Pure type systems with subtyping. In J.-Y. Girard, editor, *Typed Lambda Calculus and Applications (TLCA)*, pages 381–396. Springer-Verlag, 1999. Lecture Notes in Computer Science, volume 1581.

Абельсон, Харольд and Джеральд Джей Сассман. *Структура и интерпретация компьютерных программ*. М.: Добросвет, 2006. ISBN 5982271918.

Арнольд, Кен and Джеймс Гослинг. *Язык программирования JAVA*. Питер, 1997.

Ахо, Альфред В., Моника С. Лам, Рави Сети, and Джеффри Д. Ульман. *Компиляторы. Принципы, технологии и инструментарий*. Вильямс, 2008. ISBN 9785845913494. 2-е изд.

Керниган, Брайан and Деннис Ритчи. *Язык программирования C*. Вильямс, 2009. ISBN 9785845908919.

Митчелл, Джон К. *Основания языков программирования*. НИЦ "Регулярная и хаотическая динамика" 2010. ISBN 978-5-93972-757-0.

Страуструп, Бьорн. *Язык программирования C++*. Специальное издание. Бином, Невский Диалект, 2008.

Секрет творчества заключается
в умении скрывать источники.

Альберт Эйнштейн

Список иллюстраций

1. Зависимости между главами	14
2. Примерная схема продвинутого аспирантского курса	16
1.1. Краткая история типов в информатике и логике.	33
3.1. Булевские выражения (B)	54
3.2. Арифметические выражения (NB)	60
5.1. «Внутренний цикл» функции предшествования.	82
5.2. Вычисление <code>factorial c3</code>	86
5.3. Бестиповое лямбда-исчисление (λ)	91
8.1. Правила типизации для булевых значений (B).	111
8.2. Правила типизации для чисел (NB).	112
9.1. Чистое простое типизированное лямбда-исчисление (λ_{\rightarrow})	123
11.1. Неинтерпретируемые базовые типы	138
11.2. Единичный тип	139
11.3. Приписывание типа	142
11.4. Связывание <code>let</code>	144
11.5. Пары	146
11.6. Кортежи	148
11.7. Записи	149
11.8. (Бестиповые) образцы для записей	151
11.9. Типы-суммы	164
11.10. Суммы (с единственностью типизации)	165
11.11. Варианты	166
11.12. Рекурсия общего вида	167
11.13. Списки	168
13.1. Ссылки	191
13.2. Ссылки (<i>продолжение</i>)	192
14.1. Ошибки	194
14.2. Обработка ошибок	196
14.3. Исключения, сопровождаемые значениями	197

15.1. Простое типизированное лямбда-исчисление с подтипами ($\lambda_{<:}$)	210
15.2. Записи (совпадает с рис. 11.7)	211
15.3. Записи и подтипы	212
15.4. Наименьший тип	216
15.5. Варианты и подтипы	222
16.1. Отношение потипирования для записей (компактная версия)	237
16.2. Алгоритмическое отношение подтипирования	238
16.3. Алгоритмическая типизация	244
19.1. Облегченная Java (синтаксис и подтипы)	283
19.2. Облегченная Java (вспомогательные определения)	285
19.3. Облегченная Java (вычисление)	286
19.4. Облегченная Java (тиปизация)	287
20.1. Изорекурсивные типы ($\lambda\mu$)	307
21.1. Примеры древовидных типов.	316
21.2. Пример функции <i>support</i>	322
21.3. Пример применения функции <i>treeof</i>	331
21.4. Конкретный алгоритм проверки подтипирования для μ -типов	335
21.5. Алгоритм проверки подтипирования Амадио и Карделли	340
22.1. Правила типизации с ограничениями	351
22.2. Алгоритм унификации	357
23.1. Полиморфное лямбда-исчисление (Система F)	391
24.1. Экзистенциальные типы	398
26.1. Ограниченнная квантификация (ядерная $F_{<:}$)	422
26.2. «Полная» ограниченная квантификация	425
26.3. Ограниченные кванторы существования (ядерный вариант)	436
28.1. Алгоритм выявления для $F_{<:}$	448
28.2. Алгоритмическая типизация для $F_{<:}$	449
28.3. Алгоритмическое отношение подтипирования для ядерной $F_{<:}$	452
28.4. Алгоритмическое отношение подтипирования для полной $F_{<:}$	454
28.5. Алгоритмы поиска объединений и пересечений для ядерной $F_{<:}$	462
29.1. Операторы над типами и виды (λ_ω)	478
30.1. Полиморфное лямбда-исчисление высших порядков (F_ω)	495
30.1. Полиморфное лямбда-исчисление высших порядков (F_ω) (<i>продолжение</i>)	496
30.2. Экзистенциальные типы высших порядков	497
30.3. Параллельная редукция на типах	497
31.1. Ограниченнная квантификация высших порядков ($F_{<:}^\omega$)	505

31.1. Ограниченнaя квантificationa высших порядков ($F_{\zeta,:}^\omega$) (<i>продолжение</i>)	506
32.1. Полиморфное обновление	516
A.1. Типизированные образцы для записей	541
A.2. Наименьший надтип и наибольший подтип данного типа, не содержащие X	588

Предметный указатель

- АСД, *см.* дерево, абстрактное синтаксическое
ATD, *см.* тип данных, абстрактный
Пенсильванская трансляция, 229
Система $F_<$:
 полная, 421
 ядерная, 421
Система F , 369, 371
абстрагирование, 219
абстракция, 143
 определеняемая пользователем, 23,
 369
 полная, 161
абстракция данных
 объектно-ориентированная, 402
абстракция типа, 372
абстрактные типы данных, *см.* тип
 данных, абстрактный
аксиома, 47
алгоритм
 унификации типов, 357
 W, 366
альфа-конверсия, 90
анализ
 размерностей, 25
 типов
 консервативный, 110
анализатор
 грамматический, 73
 лексический, 73
аннотация типов, 24
 явная, 154
ассоциативность операторов, 73
барьер абстракции, 399
бета-редукция, 75
 полная, 76
безопасность, 23
библиотека
 обобщенная, 361
бисимуляция, 314
число
 Чёрча, 80, 377
 натуральное, 35
 теговое представление, 226
 упакованное представление, 226
делегация, 292
делегат, 255
дерево, *см.* тип, древовидный
 абстрактное синтаксическое, 73
 вывода, 113
 вывода типов, 112, 408
дерево вывода, 56, 123
деструктор типа, 520
доказательства
 программа проверки, *см.* про-
 грамма проверки доказа-
 тельств
доказательство, 492
 через логические отношения, 170
 индуктивное, 49
 сценарий, 492
домен
 семантический, 53
 универсальный, 303
эффект
 вычислительный, 175
эквивалентность определений, 471,
 476
финализатор, 546
форма
 каноническая, 114
 нормальная, 57, 84

- производная, 71, 137, 140
- формальный метод, 21
- функция, 36
 - частичная, 36, 67
 - генератор, 161
 - интерпретации, 53
 - мера завершения, 59
 - монотонная, 312
 - обобщенная, 371
 - определенена, 36
 - перехода (машины состояний), 52
 - перевода
 - когерентность, 230
 - порождающая, 312
 - обратимая, 320
 - раскрытия сокращений, 140
 - с конечным множеством состояний, 324
 - терпящая неудачу, 36
 - тождества, 75, 372
 - высшего порядка, 78
 - всюду определенная, 36
- функция-обёртка, 267
- глубина терма, 49
- граф абстрактного синтаксиса, 77
- грамматика
 - категориальная, 29
- граница
 - точная нижняя, см. пересечение
 - точная верхняя, см. объединение
- хвостовой вызов, см. вызов, хвостовой
- имена типов, 280
- индекс де Брауна, 94, 99, 411
- индукция
 - по выводам, 56
 - структурная, 40
 - вложенная, 40
- интерфейс, 254, 255, 289
- интерпретатор
 - equirec, 311
 - fomega, 591
 - fomsub, 499
 - fullequirec, 297, 300
 - fullerror, 193
 - fullfomsub, 419, 499
 - fullfsubref, 441
- fullfsub, 447
- fullisorec, 297, 308
- fullomega, 369, 376, 402, 404, 469
- fullpoly, 369, 393
- fullrecon, 347
- fullref, 175, 253, 260, 268
- fullsimple, 119, 137
- fullsub, 205, 419
- fulluntyped, 71, 93, 101
- fullupdate, 507
- purefsub, 447, 489
- rcdsbot, 558
- rcdsb, 205, 251
- recon, 347
- untyped, 101
- инвариант, 53
- исчисление
 - Милнера-Майкрофта, 368
 - объектов, 71, 209, 292
 - операций над записями, 233
 - построений, 493
- исключение
 - порядок наследования, 200
 - повторный выброс, 199
 - в Java, 200
- исполнение
 - последовательное, 139, 177
- изоморфизм Карри-Говарда, см. соотношение Карри-Говарда
- карринг, см. каррирование
- каррирование, 78, 470
- класс, 255, 259, 513
 - расширение, 259, 262
 - создание экземпляра, 259
 - внутренний, 276
- класс типов, 367
- класс-примесь, 292
- код
 - сопровождение, 25
- кодирование, 275
 - по Чёрчу, 376
- коиндукция, 311
- комбинатор, см. терм, замкнутый
 - Y-комбинатор, 85
 - неподвижной точки, 85
 - расходящийся, 84
- компиляция, 397

- компоновка, 397
 конкретизация типа, 347, 372
 наиболее общая, 349
 конструкции
 языка
 чистые, 175
 нечистые, 175
 конструкция
 логическая, 493
 конструктор
 инвариантный, 222
 конструктор объектов, 278
 конструктор типа, 163, 469, 480
 контекст именования, 95
 контекст типизации, 121
 правильно построенный, 487
 контекст вычисления, 290
 кортеж, 145, 147
 гибкий, 427
 куб Барендрегта, *см.* лямбда-куб
 куча, 175
 квантификация
 F-ограниченная, 423
 ограниченная, 419, 421
 полностью ограниченная, 461
 высшего порядка, 470
 квантор
 высших порядков
 ограниченный, 501
 лексема, 73
 лексер, *см.* анализатор, лексический
 лемма
 о подстановке, 124
 о порождении, 112
 о сужении, 431, 455
 об инверсии, 124, 135, 484
 логика
 комбинаторная, 94
 линейная, 129
 модальная, 129
 лямбда-абстракция, 73
 неявно аннотированная, 362
 лямбда-исчисление, 71, 72
 бестиповое, 71
 полиморфное, 372
 простое типизированное
 чистое, 122
 с операторами над типами, 476
 вырожденное, 123
 второго порядка, 372, 489
 лямбда-куб, 494
 лямбда-терм, 73
 машина
 абстрактная, 52, 182
 метаматематика, 44
 метапеременная, 44
 метатеория, 44
 метаязык, 44
 метка поля, 149
 метод, 254, 256
 динамическая диспетчеризация, 254
 переопределение, 261, 292
 вызов, *см.* вызов метода
 мини-ML, 367
 множество
 F-консistentное, 312
 F-замкнутое, 312
 достижимость элементов, 324
 кандидатов на редуцируемость, 170
 кумулятивное, 47
 насыщенное, 170
 ограничений, 351
 решение, 353
 унификация, 351
 порождающее, 320
 счетное, 35
 универсальное, 312
 вполне упорядоченное, 38
 модель
 программа проверки, *см.* проверка модели, программа
 модель объектов
 экзистенциальная, 507
 модуль, 394
 модульность, 23, 373
 мониторинг во время исполнения, 21
 монотип, 388
 мультиметод, 254, 293, 370
 μ -тип
 подвыражение
 при взгляде снизу вверх, 337
 при взгляде сверху вниз, 335

- μ -тип, 330
- μ -тип
 - сырой, 330
- надтип, 218
- наследование
 - множественное, 520
 - неподвижная точка (функции), 312
 - независимость представлений, 401
 - нормализация, 382
 - сильная, 382
 - строгая, 172
 - вычислением, 173
 - объединение, 37, 236, 246, 462
 - объект, 179, 256, 398
 - область видимости переменной, 75
 - обновление
 - полиморфное, 507, 511, 514
 - обобщение тела `let`, 387
 - обработчик исключений, 193, 196
 - образец, 150
 - сопоставление с, 150
 - ограничение, 373
 - окружение, 105
 - операция
 - бинарная, 405
 - сильная бинарная, 406
 - слабая бинарная, 405
 - оператор
 - перегрузка
 - конечная, 231
 - оператор над типами, 159, 297, 469, 480
 - контравариантный, 503
 - ковариантный, 503
 - ограниченный, 500, 503
 - полярность, 503
 - высших порядков, 472
 - определение
 - индуктивное, 49
 - ошибка
 - в Java, 200
 - времени выполнения, 58, 61
 - диагностируемая, 27
 - недиагностируемая, 27
 - ошибки типов
 - времени выполнения, 23
 - отношение, 35
- антисимметричное, 37
- бинарное, 36
 - рефлексивное, 37
 - сохранение предиката, 36
- эквивалентности, 37
- логическое, 169
- область значений, 36
- область определения, 36
- симметричное, 37
- типовизация, 111
 - с ограничениями, 351
- транзитивное, 37, 317
- вычисления, 54
 - многошагового, 58
 - одношаговое, 56
- отношение подтипование
 - контравариантность, 209
- отношение подтипования
 - алгоритмическое, 236, 239
 - эквивалентность типов, 223
 - ковариантность, 209
 - объединение, 246
 - пересечение, 246
 - ограниченное, 246
 - разрешающая процедура, 239
- отношение типизации
 - алгоритмическое, 236, 243
- отношение видообразования, 470
- отправка сообщения, см. вызов метода
 - да
- пакет, 394
- память, 175
 - адрес, 181
 - цикл, 185
 - освобождение, 180
 - состояние, 365
 - правильно типизированное, 187
 - структура типизации, 185
 - ячейка, 177
- пара, 145
- параметрические типы данных, см. тип данных, параметрический
- параметричность, 389
- парсер, см. анализатор, грамматический
- перегрузка функций, 370

- динамическое разрешение, 370
 статическое разрешение, 370
 переменная, 94
 строчная, 367, 575
 свободная, 88
 минимальные предположения,
 367
 свободное вхождение, 75
 связанная, 75
 типовая, 134, 347
 обобщение, 363
 слабая, 366
 захват переменных, 89
 self, 255
 this, 255
 переменная экземпляра, 256
 пересечение, 37, 246
 перестановка, 39
 пи-исчисление, 71
 подынтерфейс, 512
 подкласс, 255, 259
 подстановка
 более обобщенная, 356
 менее конкретная, 356
 проверка на вхождение, 357, 368
 свободная от захвата, 89
 типов, 348
 типовной переменной, 347
 явная, 94, 105
 подтип, 206
 подтипы, 205
 ограничение на, 421
 подтипование
 позитивное, 520
 в глубину, 207
 в ширину, 207
 ядерное правило, 424
 подтипование интерфейсов, 512
 подтипование в глубину
 метка изменчивости, 515
 полиморфизм, 360
 через подтипы, 205, 371
 через let, 361, 370
 импредикативный, 367, 370, 389
 интенсиональный, 371
 параметрический, 349, 370
 первого класса, 370
 по Дамасу-Милнеру, 361
 предикативный, 389
 ранга 2, 367, 388
 специализированный, 370
 строчных переменных, 233, 520
 в стиле ML, 361, 370
 политип, 388
 порождение кода во время выполнения, 129
 порядок
 частичный, 37
 лексикографический, 39
 линейный, 37
 последовательность, 38
 правило
 алгоритмическое
 корректность, 238
 полнота, 238
 исключенного третьего, 129
 конкретное, 47
 рабочее, 55
 соответствия, 55
 тиปизации, 113
 устранения, 128, см. Т-APP
 выполнимость на отношении, 55
 включения, 206
 введения, 128, см. Т-ABS
 B-IFFALSE, 62
 B-IFTRUE, 62
 B-ISZEROFALSE, 63
 B-ISZEROTRUE, 63
 B-PREDSUCC, 63
 B-PREDZERO, 63
 B-SUCC, 63
 B-VALUE, 62
 CT-ABSINF, 360
 CT-Abs, 351, 572
 CT-APP, 351, 572
 CT-FALSE, 351
 CT-FIX, 573
 CT-IF, 351
 CT-ISZERO, 351
 CT-LETPOLY, 362
 CT-PRED, 351
 CT-PROJ, 575
 CT-SUCC, 351
 CT-TRUE, 351

- CT-VAR, 351, 572
 CT-ZERO, 351
 E-ABS, 534
 E-APP1, 91, 123, 182, 191, 210, 391, 422, 478, 495, 505, 534, 535
 E-APP2, 91, 123, 182, 191, 210, 391, 422, 478, 495, 505, 534
 E-APPABS, 91, 99, 123, 182, 191, 210, 372, 391, 422, 478, 495, 505, 534, 535
 E-APPERR1, 194
 E-APPERR2, 194
 E-APPRAISE1, 197
 E-APPRAISE2, 197
 E-ASCRIBE1, 142
 E-ASCRIBEEAGER, 143
 E-ASCRIBE, 142, 219
 E-ASSIGN1, 183, 191
 E-ASSIGN2, 183, 191
 E-ASSIGN, 183, 191
 E-CASEINL, 164, 165
 E-CASEINR, 164, 165
 E-CASEVARIANT, 166
 E-CASE, 164, 166
 E-CASTNEW, 286
 E-CAST, 286
 E-CONS1, 168
 E-CONS2, 168
 E-DEREFLOC, 183, 191
 E-DEREF, 183, 191
 E-DOWNCAST, 219
 E-FIELD, 286
 E-FIXBETA, 167
 E-FIX, 167
 E-FLD, 307
 E-GC, 545
 E-HEADCONS, 168
 E-HEAD, 168
 E-IF-WRONG, 62
 E-IFFALSE, 54
 E-IFTRUE, 54
 E-IF, 54
 E-INL, 164, 165
 E-INR, 164, 165
 E-INVK-ARG, 286
 E-INVK-RECV, 286
 E-INVKNEW, 286
 E-IsZERO-WRONG, 62
 E-IsZERO, 60
 E-IsnILCons, 168
 E-IsnNIL, 168
 E-IsnIL, 168
 E-IsZEROSUCC, 60
 E-IsZEROZERO, 60
 E-LETV, 144, 151, 362
 E-LET, 144, 151, 365
 E-NEW-ARG, 286
 E-PACK, 398, 497
 E-PAIR1, 146
 E-PAIR2, 146
 E-PAIRBETA1, 146
 E-PAIRBETA2, 146
 E-PRED-WRONG, 62
 E-PREDSUCC, 60, 68
 E-PREDZERO, 60
 E-PRED, 60
 E-PROJ1, 146
 E-PROJ2, 146
 E-PROJNEW, 286
 E-PROJRCD, 149, 211, 226, 516
 E-PROJTUPLE, 148
 E-PROJ, 148, 149, 211
 E-RAISERAISE, 197
 E-RAISE, 197
 E-RCD, 149, 211, 516
 E-REFV, 183, 191
 E-REF, 183, 191
 E-SEQNEXT, 140
 E-SEQ, 139
 E-SUCC-WRONG, 62
 E-SUCC, 60
 E-TAPP, 391, 422, 495, 505
 E-TAILCONS, 168
 E-TAIL, 168
 E-TAPPTABS, 372, 391, 415, 495, 505
 E-TRYERROR, 196
 E-TRYRAISE, 197
 E-TRYV, 196, 197
 E-TRY, 196, 197
 E-TUPLE, 148
 E-TYPETEST1, 220
 E-TYPETEST2, 220

- E-UNFLDFLD, 307
 E-UNFLD, 307
 E-UNPACKPACK, 397, 398, 497
 E-UNPACK, 398
 E-UPDATEV, 516
 E-VARIANT, 166
 E-WILDCARD, 539
 K-ABS, 478, 495, 505
 K-ALL, 495, 505
 K-APP, 478, 495, 505
 K-ARROW, 478, 495, 505
 K-SOME, 497
 K-TVAR, 478, 495, 505
 K-TOP, 505
 M-RCD, 151
 M-VAR, 151
 P-RCD', 541
 P-RCD, 541
 P-VAR, 541
 Q-ABS, 478, 496, 506
 Q-ALL, 496, 506
 Q-APPABS, 471, 478, 496, 506
 Q-APP, 478, 496, 506
 Q-ARROW, 478, 496, 506
 Q-REFL, 478, 496, 506
 Q-SOME, 497
 Q-SYMM, 478, 496, 506
 Q-TRANS, 478, 496, 506
 QR-ABS, 497
 QR-ALL, 497
 QR-APPABS, 497
 QR-APP, 497
 QR-ARROW, 497
 QR-REFL, 497
 S-ABS, 500, 506
 S-ALL, 422, 424, 425, 457, 506
 S-AMBER, 341
 S-APP, 500, 506
 S-ARRAYJAVA, 223
 S-ARRAY, 223
 S-ARROW, 209, 210, 237, 422, 506
 S-ASSUMPTION, 341
 S-BOT, 216
 S-EQ, 500, 506
 S-INTER1, 231
 S-INTER2, 231
 S-INTER3, 231
 S-INTER4, 231
 S-LIST, 222
 S-PRODDEPTH, 212
 S-PRODWIDTH, 212
 S-RCDDEPTH, 207, 212, 516
 S-RCDPERM, 208, 212
 S-RCDVARIANCE, 516
 S-RCDWIDTH, 207, 212, 516
 S-RCD, 237
 S-REFSINK, 224
 S-REFSOURCE, 224
 S-REFL, 207, 210, 237, 422
 S-REF, 223
 S-SINK, 224
 S-SOME, 436, 508, 585
 S-SOURCE, 224
 S-TVAR, 422, 424, 506
 S-TOP, 209, 210, 237, 422, 506
 S-TRANS, 207, 210, 235, 237, 422,
 506
 S-VARIANTDEPTH, 222
 S-VARIANTPERM, 222
 S-VARIANTWIDTH, 222
 SA-ALL, 452, 454
 SA-ARROW, 238, 452, 454
 SA-BOT, 247
 SA-RCD, 238
 SA-REFL-TVAR, 452, 454
 SA-TOP, 238, 452, 454
 SA-TRANS-TVAR, 452, 454
 T-ABS, 121–123, 192, 210, 391,
 422, 478, 496, 506
 T-APP, 122, 123, 192, 206, 210,
 391, 422, 478, 496, 506
 T-ASCRIBE, 142, 218
 T-ASSIGN, 180, 187, 192, 224
 T-CASE, 164, 166
 T-CAST, 561
 T-CONS, 168
 T-DCAST, 287
 T-DEREF, 180, 187, 192, 224
 T-DOWNCAST, 219
 T-EQ, 471, 478, 496
 T-ERROR, 194
 T-EXN, 197
 T-FALSE, 111
 T-FIELD, 287

- T-FIX, 167
- T-FLD, 307
- T-HEAD, 168
- T-IF, 111, 122, 245
- T-INL, 164, 165
- T-INR, 164, 165
- T-INVK, 287
- T-ISZERO, 112
- T-ISNIL, 168
- T-LETPOLY, 362, 363
- T-LET, 144, 362, 541
- T-LOC, 186, 192
- T-NEW, 287
- T-NIL, 168
- T-PACK, 395, 398, 436, 497
- T-PAIR, 146
- T-PRED, 112
- T-PROJ1, 146
- T-PROJ2, 146
- T-PROJ, 148, 149, 211, 516
- T-RCD, 149, 211, 516
- T-REF, 180, 186, 192
- T-SCAST, 287
- T-SEQ, 140
- T-SUB, 206, 210, 235, 422, 506
- T-SUCC, 112
- T-TABS, 372, 391, 422, 425, 496, 506
- T-TAPP, 373, 391, 422, 425, 496, 506
- T-TAIL, 168
- T-TAPPTABS, 422
- T-TRUE, 111
- T-TRY, 196, 197
- T-TUPLE, 148
- T-TYPETEST, 220
- T-UCAST, 287
- T-UNFLD, 307
- T-UNIT, 139, 192
- T-UNPACK, 396, 398, 436, 465, 497
- T-UPDATE, 516
- T-VARIANT, 166, 222
- T-VAR, 122, 123, 192, 210, 287, 391, 422, 478, 496, 506
- T-WILDCARD, 539
- T-ZERO, 112
- TA-ABS, 244, 449
- TA-APPBOT, 247
- TA-APP, 244, 449
- TA-IFD, 557
- TA-IF, 246, 557
- TA-PROJBOT, 247
- TA-PROJ, 244
- TA-RCD, 244
- TA-TABS, 449
- TA-TAPP, 449
- TA-UNPACK, 466
- TA-VAR, 244, 449
- XA-OTHER, 448
- XA-PROMOTE, 448
- правило подтипования
 - полное, 421
 - ядерное, 421
- правило типизации
 - ограничение на значения, 365, 387
- правило вывода, 46, 313
 - экземпляр, 55
- предикат, 35, см. также отношение, одноместное
 - логический, 170
 - сохранение отношением, 36
- предпорядок, 37, 209
 - полный, 38
 - убывающая цепочка, 38
- пренекс-полиморфизм, 387
- применение типа, 372
- принцип
 - безопасной подстановки, 206
- приоритет операторов, 73
- присваивание, 176
- присвоение видов, 476
- приведение типов, 218, 278, 386
 - «глупое», 287
 - предупреждение, 288
 - находящее, 27, 218, 287
 - восходящее, 218, 287
- процесс-функция, 301
- продолжение, 407
- проектирование абстракций, 402
- программа
 - проверки доказательств, 24
 - проверки моделей, см. проверка модели, программа

- проверка моделей, 314
 программа, 21
- проверка типов
 статическая, 110
- проверка выхода за границы массива, 27
- псевдоним, 178
 анализ, 190
 паразитные взаимодействия, 190
- расхождение, 36
- расширение типа, 448
- размер терма, 49, 88
- редекс, 75
- редукция, 54, 488
 параллельная, 482
 слабая заголовочная, 488
- рефлексия, в Java, 221
- реконструкция типов, 121, 131, 343, 347, 349, 370, 383, 537
 частичная, 367
 локальная, 384
- рекурсия
 общего вида, 263
 полиморфная, 368
- решение
 главное, 359
- сборка мусора, 180
- семантика, 52
 аксиоматическая, 53
 денотационная, 52
 естественная, 52, 53, 62
 операционная, 52
 с большим шагом, 52, 62
 с малым шагом, 52, 62
 структурная, 52, 53
- подтипы
 на основе подмножеств, 206
 на основе преобразования типов, 225, 252, 408
 с передачей типов, 386
 со стиранием типов, 386
- схема правил, 47
- схема типа, 363
 конкретизация, 364
- схема типов, 388
- синтаксический сахар, 141
 удаление, 141
- синтаксис, см. также синтаксический сахар
 абстрактный, 45, 73
 конкретный, 73
- система линейных типов, 129
- система модулей, 409, 493
- система присвоения типов, 121
- система типов, 21
 безопасность, 113
 чистая, 22
 чистых, 475, 494
 именная, 280, 298
 консервативная, 23
 корректность, 23, 113
 полиморфная, 370
 структурная, 280, 298
- смысл терма, 52
- сообщение, отправка, см. вызов метода
- соотношение Карри-Говарда, 22, 128, 372, 459
- соответствие, 512
- состояние
 абстрактной машины, 52
- состояние объекта, 256
- ссылка, 176, 177, 181, 376
 присваивание, 176
 разыменование, 176
 выделение памяти, 176
 висячая, 180
- степенной вид, 439
- стиль Чёрча, 131
- стиль Карри, 131
- стирание типов, 130, 382
- стратегия вычисления, 55
 нестрогая, 77
- свойство
 расширения субъекта, 128
- свойство Чёрча-Россера, 482
- связывание
 позднее, 255, 263
- связывание-пустышка, 141
- связывающее определение, 75
- тактика доказательств, 492
- тег
 постановка тегов, 152

- тег типа, 22, 221
 теорема
 о единственности типов, 125
 о продвижении, 113
 о редукции субъекта, 116
 о сохранении, 113
 теория доменов, 53
 теория категорий, 31
 теория типов, см. система типов
 конструктивная, 22
 простая, 22
 теория типов Рассела, 22
 терм, 44
 безымянный, см. терм де Брауна
 де Брауна, 94
 эквивалентность поведенческая, 84
 корректно типизированный, 111
 незашщенный, 385
 нормализуемый, 169, 231
 новое имя, 140
 определение множества
 через правила вывода, 46
 порождение элементов, 47
 расходящийся, 84
 тиปизируемый, 131, 383
 тупиковый, 61
 замкнутый, 75
 терм `error`
 тип, 195
 терм, замкнутый, см. также комбинатор
 терм-значение, 45, 54, 77
 тип, 472
 абстрактный, 500
 атомарный, 138
 базовый, 137, 163
 неинтерпретируемый, 137, 347
 неизвестный, 137
 частично абстрактный, 437
 декартово произведение, 146
 динамическая проверка, 218
 древовидный, 315
 конечный, 315
 поддерево, 329
 регулярный, 329
 эквивалентность типов, 150
 экзистенциальный
 логическая интуиция, 393
 ограниченный, 436
 операционная интуиция, 394
 полиморфная перепаковка, 520
 введение, 394
 главный, 347, 359
 каналов взаимодействия, 225
 конструктор типов, 120
 наименьший, 245, 447
 неопределенный элемент, 162
 непересекающееся объединение, 159
 объявление перед использованием, 158
 ограничений, 367
 параметризованный, 159
 представления, 258
 преобразование, 143, 205
 приписывание, 141, 218
 простой, 120, 472
 распечатка, 141
 регулярный, 329
 реконструкция, см. реконструкция типов
 рекурсивный, 152, 159, 180, 281, 297
 эквирекурсивное представление, 298, 305
 изорекурсивное представление, 298, 306, 341
 структурное развертывание, 520
 сырой, 276
 сократимый, 330
 специфицированный, 367
 степенной, 476
 универсальный
 логическая интуиция, 393
 операционная интуиция, 393
 вариантный
 расширяемый, 199
 зависимый, 27, 490
 зависимый функциональный, 491
`Bot`, 195
`Dynamis`, 160
 тип данных

- абстрактный, 254, 398, 399
 параметрический, 475
 разнородный, 152
 тип форм, 493
 тип состояния, 507
 тип терма
 композициональность вычисле-
 ния, 22
 тип-объединение, 160, 232
 тип-перечисление, 156
 тип-пересечение, 231, 388, 430, 520
 тип-представление, 394
 тип-произведение, 146
 тип-сумма, 152
 тип-свидетель, 394
 тип-уточнение, 232
 тип-вариант, 152, 154, 198
 типизация
 динамическая, 22
 главная, 367
 латентная, 22
 типизируемый терм, *см.* терм, кор-
 ректно типизированный
 тождество объектов, 272
 указатель, 176, 181
 арифметика с ними, 181
 слабый, 546
 унификация, 356
 эквациональная, 367, 576
 унификанд
 экзистенциальный, 350
 унификатор
 главный, 356
 наиболее общий, 356
 универсум, *см.* множество, универ-
 сальное
 уровень де Брауна, 99
 устранение сечений, 129
 метод доказательства, 318
 утверждение о вычислении
 выводимость, 56
 утверждения о типах, 113
 вес типа, 453
 вычисление, 54
 частичное, 129
 управляемое типами, 173
 нормальный порядок, 76
 субъекта, 116
 выделение, 35
 вывод регионов, 28
 вывод типов, 121, 350
 локальный, 384
 окрашенный, 384
 жадный, 384
 вызов
 хвостовой, 326
 метода, 254, 278
 по имени, 76
 по необходимости, 77
 по значению, 77
 вид, 373
 одноэлементный, 471, 476, 493
 степенной, 476
 строчный, 476
 уровня *i*, 489
 зависимый, 476
 вид типа, 471
 вид записей, 475
 ядерная Fun, 424
 янтарное правило, 341
 язык
 неявно типизированный, 121
 объектный, 44
 объектно-ориентированный, 205
 описания модулей, 25
 промежуточный, 182
 внешний, 73
 внутренний, 73
 явно типизированный, 121
 Amber, 341
 Cecil, 254
 CLOS, 254
 Dylan, 254
 Java, 371
 минимальное ядро, 276
 KEA, 254
 MzScheme, 397
 Objective Caml, 65
 Smalltalk, 254
 замыкание
 рефлексивно-транзитивное, 37
 рефлексивное, 37
 транзитивное, 37
 запись активации, 195

- значение
 числовое, 60
 необязательное, 155
- let**-форма
 обобщение полиморфных определений, 145
- 0-терм, 95
- abstract data type, см. тип данных, абстрактный
- abstract machine, см. машина, абстрактная
- abstract syntax, см. синтаксис, абстрактный
- abstract syntax tree, см. дерево, абстрактное синтаксическое
- abstraction, см. абстрагирование
- ad-hoc polymorphism, см. полиморфизм, специализированный
- ADT, см. тип данных, абстрактный
- algorithmic subtyping, см. отношение подтипов, алгоритмическое
- algorithmic typing, см. отношение типизации, алгоритмическое
- algorithmic typing relation, см. отношение типизации, алгоритмическое
- Algorithm W, см. алгоритм W
- alias, см. псевдоним
- alias analysis, см. псевдоним, анализ
- alpha-conversion, см. альфа-конверсия
- Amber rule, см. янтарное правило
- array bounds checking, см. проверка выхода за границы массива
- arrow type, см. тип, функциональный
- ascription, см. тип, присваивание
- assignment, см. присваивание
- AST, см. abstract syntax tree
- axiom, см. аксиома
- axiomatic semantics, см. семантика, аксиоматическая
- Barendregt cube, см. лямбда-куб
- beta-reduction, см. бета-редукция
- big-step operational semantics, см. семантика операционная с большим шагом
- bisimulation, см. бисимуляция
- bottom-up subexpression, см. тип, подвыражение, при взгляде снизу вверх
- bound variable, см. переменная, связанная
- bounded existential, см. тип, экзистенциальный ограниченный
- bounded quantification, см. квантификация, ограниченная
- bounds, см. ограничения
- calculus of constructions, см. исчисление построений
- call
 by name, см. вызов по имени
 by need, см. вызов по необходимости
 by value, см. вызов по значению
- call-by-value Y-combinator, см. Y-комбинатор, с вызовом по значению
- canonical form, см. форма, каноническая
- capture-avoiding substitution, см. подстановка, свободная от захвата
- casting, см. приведение типов, см. тип, преобразование
- categorial grammar, см. грамматика, категориальная
- Church encoding, см. кодирование по Чёрчу
- Church numeral, см. число Чёрча
- Church-style, см. стиль Чёрча
- class, см. класс
- closed term, см. терм, замкнутый
- coercion, см. преобразование типов
- coercion semantics, см. семантика подтипов на основе преобразования типов
- combinator, см. терм, замкнутый
- combinatory logic, см. логика, комбинаторная
- completely bounded quantification, см. квантификация, полностью ограниченная

- comprehension, см. выделение
 computation rule, см. правило, рабочее
 computational effect, см. эффект, вычислительный
 concrete rule, см. правило, конкретное
 concrete syntax, см. синтаксис, конкретный
 congruence rule, см. правило, соответствия
cons с хэшированием, 250
 constraint set, см. множество ограничений
 constraint type, см. тип ограничений
 continuation, см. продолжение
 Curry-Howard correspondence, см. соотношение Карри-Говарда
 Curry-Howard isomorphism, см. соотношение Карри-Говарда
 Curry-style, см. стиль Карри
 currying, см. каррирование
 cut elimination, см. устранение сечений
 proof, см. устранение сечений, метод доказательства
 dangling reference, см. ссылка, висячая
 de Bruijn
 index, см. индекс де Брауна
 level, см. уровень де Брауна
 term, см. терм де Брауна
 decision procedure, см. отношение подтиповирования, разрешающая процедура
 denotational semantics, см. семантика, денотационная
 dependent function types, см. тип, зависимый функциональный
 dependent kind, см. вид, зависимый
 depth subtyping, см. подтиповирование, в глубину
 derived form, см. форма, производная
 desugaring, см. синтаксический сахар, удаление
 dimension analysis, см. анализ размерностей
 domain theory, см. теория доменов
 down-cast, см. преобразование типов, нисходящее, см. приведение типов, нисходящее
 encoding, см. кодирование
 enumeration, см. тип-перечисление
 environment, см. окружение
 equational unification, см. унификация, эквациональная
 equivalence relation, см. отношение эквивалентности
 erasure, см. стирание типов
 evaluation, см. вычисление
 evaluation strategy, см. стратегия вычисления
 exception handler, см. обработчик исключений
 existential unifcands, см. унификанд, экзистенциальный
 external language, см. язык, внешний
F-bounded quantification, см. квантификация, *F*-ограниченная
F-closed set, см. множество, *F*-закрытое
F-consistent set, см. множество, *F*-консистентное
 finalizer, см. финализатор
 finitary overloading, см. оператор, перегрузка, конечная
 first-class polymorphism, см. полиморфизм, первого класса
 flexible tuple, см. кортеж, гибкий
 formal method, см. формальный метод
 free variable, см. переменная, свободная
 function, см. функция
 garbage collection, см. сборка мусора
 general recursion, см. рекурсия, общего вида
 generation lemma, см. лемма, о порождении
 generics, см. функция, обобщенная
 hash consing, см. **cons** с хэшированием

- higher-order function, см. функция, высшего порядка
- higher-order quantification, см. квантификация, высшего порядка
- higher-order type operator, см. оператор над типами, высших порядков
- identity function, см. функция тождества
- impredicative polymorphism, см. полиморфизм, импредикативный
- inference rule, см. правило вывода
- inner induction, см. индукция, вложенная
- instantiation, см. конкретизация
- intensional polymorphism, см. полиморфизм, интенсиональный
- interface, см. интерфейс
- intermediate language, см. язык, промежуточный
- internal language, см. язык, внутренний
- interpretation function, см. функция интерпретации
- intersection type, см. тип-пересечение
- inversion lemma, см. лемма, об инверсии
- join, см. объединение
- Kernel Fun, см. ядерная Fun
- kind, см. вид типа
- lambda-calculus, см. лямбда-исчисление
- lambda-cube, см. лямбда-куб
- lambda-term, см. лямбда-терм
- late binding, см. позднее связывание
- let-polymorphism, см. полиморфизм, через let
- lexer, см. анализатор, лексический
- lexical analyzer, см. анализатор, лексический
- LF, 493
- meet, см. пересечение
- message, sending, см. вызов метода
- metalanguage, см. метаязык
- metamathematics, см. метаматематика
- metatheory, см. метатеория
- metavariable, см. метапеременная
- method invocation, см. вызов метода
- Milner-Mycroft calculus, см. исчисление Милнера-Майкрофта
- mini-ML, см. мини-ML
- minimal assumptions, см. переменная, свободная, минимальные предположения
- model checker, см. проверка модели, программа
- modularity, см. модульность
- module, см. модуль
- μ -type, см. μ -тип
- multi-method, см. мульти-метод
- multiple inheritance, см. наследование, множественное
- nameless term, см. терм де Брауна
- naming context, см. контекст именования
- natural semantics, см. семантика, естественная
- normalizable term, см. терм, нормализуемый
- object, см. объект
- object calculus, см. исчисление объектов
- object language, см. язык, объектный
- object-oriented language, см. язык, объектно-ориентированный
- Objective Caml, 65
- OCaml, см. Objective Caml
- operational semantics, см. семантика, операционная
- package, см. пакет
- pair, см. пара
- parametric polymorphism, см. полиморфизм, параметрический
- parametricity, см. параметричность
- parser, см. анализатор, грамматический

- partial evaluation, см. вычисление, частичное
- partial function, см. функция, частичная
- partially abstract type, см. тип, частично абстрактный
- pattern, см. образец
- pattern matching, см. образец, сопоставление с
- PCF, 161
- Penn translation, см. Пенсильванская трансляция
- pi-calculus, см. пи-исчисление
- pointer arithmetic, см. указатели, арифметика с ними
- polymorphic lambda-calculus, см. лямбда-исчисление, полиморфное
- polymorphic recursion, см. рекурсия, полиморфная
- polymorphic update, см. обновление, полиморфное
- polymorphism, см. полиморфизм
- prenex-polymorphism, см. пренекс-полиморфизм
- preorder, см. предпорядок
- principal solution, см. решение, главное
- principal type, см. главный тип
- principal typing, см. типизация, главная
- principal unifier, см. унифициатор, главный
- proof-carrying code, см. код, содержащий доказательство
- pure lambda-calculus, см. лямбда-исчисление, бестиповое
- qualified type, см. тип, специфицированный
- rank-2 polymorphism, см. полиморфизм, ранга 2
- raw μ -type, см. μ -тип, сырой
- recursive type, см. тип, рекурсивный
- reduction, см. редукция
- reference, см. ссылка
- refinement type, см. тип-уточнение
- reflexive and transitive closure, см. замыкание, рефлексивно-транзитивное
- reflexive closure, см. замыкание, рефлексивное
- relation, см. отношение
- representation type, см. тип представления
- row kind, см. вид, строчный
- row variable, см. переменная, строчная
- row variable polymorphism, см. полиморфизм, строчных переменных
- rule schema, см. схема правил
- run-time error, см. ошибка, времена выполнения
- run-time monitoring, см. мониторинг во время исполнения
- safety, см. безопасность
- second-order lambda-calculus, см. лямбда-исчисление, второго порядка
- semantic domain, см. домен, семантический
- semantics, см. семантика
- sending a message, см. вызов метода
- simple theory of types, см. теория типов, простая
- small-step operational semantics, см. семантика операционная с малым шагом
- store, см. память, состояние
- store location, см. память, адрес
- store typing, см. память, структура типизации
- structural induction, см. индукция, структурная
- structural operational semantics, см. семантика операционная, структурная
- stuck term, см. терм, туниковый
- subclass, см. подкласс
- subset semantics, см. семантика подтипов на основе подмножеств
- substitution, см. подстановка

- substitution lemma, *см.* лемма, о подстановке
 subsumption, *см.* правило включения
 subtype, *см.* подтип
 subtyping, *см.* подтипы
 subtyping constraint, *см.* подтипы, ограничение на
 sum type, *см.* тип-сумма
 supertype, *см.* надтип
 surface syntax, *см.* синтаксис, конкретный
 syntactic control of interference, *см.* псевдоним, анализ, паразитные взаимодействия
 syntactic sugar, *см.* синтаксический сахар
 System F, *см.* Система F
 tail call, *см.* вызов, хвостовой
 term, *см.* терм
 term elaboration function, *см.* функция, раскрытия сокращений
 thunk, 267
 token, *см.* лексема
 top-down subexpression, *см.* тип, подвыражение, при взгляде сверху вниз
 total induction, *см.* индукция, всюду определенная
 tree, *см.* тип, древовидный
 tree type, *см.* тип, древовидный
 tuple, *см.* кортеж
 typable term, *см.* терм, корректно типизированный
 type, *см.* тип
 type abstraction, *см.* абстракция типа
 type application, *см.* применение типа
 type class, *см.* класс типов
 type constructor, *см.* тип, конструктор типов
 type erasure, *см.* стирание типов
 type inference, *см.* вывод типов
 type instantiation, *см.* конкретизация типа
 type operator, *см.* оператор над типами
 type reconstruction, *см.* реконструкция типов, *см.* вывод типов
- type substitution, *см.* подстановка типов
 type system, *см.* система типов
 type variable, *см.* переменная, типовая
 type-erasure semantics, *см.* семантика со стиранием типов
 type-passing semantics, *см.* семантика с передачей типов
 typing derivation, *см.* дерево, вывода типов
 typing relation, *см.* отношение типизации
 typing rule, *см.* правило типизации
 unification, *см.* унификация
 unification algorithm, *см.* алгоритм унификации типов
 union type, *см.* тип-объединение
 untyped lambda-calculus, *см.* лямбда-исчисление, бестиповое
 up-cast, *см.* приведение типов, восходящее
 variant type, *см.* тип, вариантный
 weak head reduction, *см.* редукция, слабая заголовочная
 weak type variable, *см.* переменная, типовая слабая
 well-typed term, *см.* терм, корректно типизированный
 width subtyping, *см.* подтипованиe, в ширину
 wildcard binder, *см.* связывание-пустышка
 witness type, *см.* тип-свидетель
- Y*-комбинатор
- с вызовом по значениюю, 85