1977

Можно ли освободить программирование от стиля фон Неймана? Функциональный стиль и соответствующая алгебра программ

Дж. Бэкус

Исследовательская лаборатория IBM, Сан-Хосе

Тьюринговская премия ACM за 1977 г. была присуждена Дж. Бэкусу на ежегодной конференции ACM в Сиэтле 17 октября. Представляя лауреата, председатель Комитета по премиям Дж. Е. Семмет сделал следующие замечания и зачитал часть заключительного представления. (Полный текст представления опубликован в выпуске Communications of ACM за сентябрь 1977 г., с. 681.)

«Вероятно, в этом зале не найдется человека, который не слыхал бы о Фортране, и большинство из вас, наверное, использовали его хотя бы однажды или по крайней мере смотрели из-за плеча кого-то, кто писал программу на Фортране. Наверное, почти столько же людей слыхали комбинацию букв БНФ, хотя не обязательно знают, что означают эти буквы. Так вот, «Б» означает Бэкуса, а остальные буквы объясняются в формальном представлении комитета. По моему мнению, эти два достижения входят в число полдюжины наиболее важных вкладов в информатику, и оба принадлежат Джону Бэкусу (который в случае Фортрана привлек к работе нескольких коллег). Именно за эти достижения он получает Тьюринговскую премию этого года».

Вкратце, он представлен к премии за «глубокий, оказавший решающее влияние и долговременный вклад в проектирование практических систем программирования высокого уровня, особенно проявившийся в его работе по Фортрану и в плодотворной публикации формальных процедур для спецификаций языков программирования».

Основная часть полного представления к премии сформулирована так:

«...Бэкус возглавлял небольшую группу IBM в Нью-Йорке в начале 50-х гг. Первым результатом работы этой группы явился язык высокого уровня для научных и технических вычислений, названный Фортраном. Та же группа разработала

первую систему для трансляции программ на Фортране в машинный код. Они применяли новые методы оптимизации для генерации быстрых программ в машинном коде. Для этого языка разрабатывалось много других компиляторов, сначала на машинах IBM, а впоследствии практически на любых компьютерах. В 1966 г. Фортран был принят как национальный стандарт США.

В конце 50-х гг. Бэкус работал в международном комитете, который разработал Алгол-58 и следующую версию Алгол-60. Язык Алгол и производные от него компиляторы получили широкое признание в Европе как средство разработки программ и как формальное средство публикации алгоритмов, на которых основываются программы.

В 1959 г. Бэкус представил на конференцию Юнеско в Париже доклад о синтаксисе и семантике предлагаемого международного алгебраического языка. В этом докладе он впервые применил формальный метод спецификации синтаксиса языков программирования. Эта формальная нотация получила известность как БНФ, что означает «Бэкуса нормальная форма» или «Бэкуса — Наура форма» в знак признания дальнейшего вклада П. Наура из Дании.

Итак, Бэкус внес значительный вклад как в прагматику решения задач на компьютерах, так и в теорию взаимосвязи между искусственными языками и вычислительной лингвистикой. Фортран остается одним из наиболее широко используемых во всем мире языков программирования. Теперь почти все языки программирования описываются с помощью некоторого вида формального определения синтаксиса».

Обычные языки программирования постоянно растут по объему, но не по своей выразительной силе. Неустранимые дефекты на самом фундаментальном уровне делают их тучными и слабыми: их примитивный стиль программирования «слово за словом» восходит к их общему прародителю — компьютеру фон Неймана; в них семантика тесно сплетается с переходами состояний; они разделяют программирование на мир выражений и мир операторов; они не дают возможности эффективно использовать мощные приемы комбинирования для построения новых программ из уже существующих, и в них недостает полезных математических свойств для рассуждений о программах.

Другой, функциональный стиль программирования находит применение в комбинационных формах создания программ. Функциональные программы оперируют со структурированными данными, часто оказываются неповторяемыми и нерекурсивными, конструируются иерархически, не именуют свои аргументы и не требуют сложной техники описаний процедур в целях универсальности. В комбинационных формах программы высокого уровня могут использоваться для построения программ еще более высокого уровня в таком стиле, который недоступен для традиционных языков.

С функциональным стилем программирования ассоциируется алгебра программ, в которой переменными служат программы, а операциями являются комбинационные формы. Эта алгебра может служить для преобразования

программ и для решения уравнений, в которых «неизвестными» являются программы, причем процесс решения во многом аналогичен преобразованиям уравнений в курсе высшей алгебры. Применяемые преобразования задаются алгебраическими законами и выполняются на том же языке, на котором пишутся программы. Комбинационные формы выбираются не только из-за их программистской выразительности, но и из-за мощи соответствующих им алгебраических законов. Общие теоремы алгебры задают детали поведения и условия завершения для больших классов программ.

В новом классе вычислительных систем стиль функционального программирования используется и в языке программирования, и в правилах перехода между состояниями. В отличие от языков фон Неймана в этих системах семантика слабо связана с состояниями — в течение большого вычисления происходит только одна смена состояний.

ВВЕДЕНИЕ

Я глубоко признателен за оказанную мне честь приглашением от ACM прочесть тьюринговскую лекцию 1977 г. и опубликовать этот отчет о ней с подробностями, обещанными в лекции. Читателям, желающим ознакомиться с резюме этой публикации, следует обратиться к последнему разделу (разд. 16).

1. ТРАДИЦИОННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ: ТУЧНОСТЬ И ВЯЛОСТЬ

По-видимому, с языками программирования происходит что-то неладное. Всякий новый язык включает с небольшими изменениями все свойства своих предшественников плюс кое-что еще. Руководства по некоторым языкам занимают более 500 страниц; в других случаях сложные описания втиснуты в менее пространные руководства с помощью неудобопонятного формализма. Министерство обороны планирует в настоящее время стандарт спроектированного комитетом языка, для которого может потребоваться руководство примерно в 1000 страниц. В каждом новом языке заявляются новые и удобные возможности, например сильная типизация или структурированные управляющие операторы, но фактически дела обстоят так, что лишь немногие языки снижают затраты на программирование или повышают его надежность в достаточной степени, чтобы оправдать стоимость создания этих языков и обучения их использованию.

Поскольку большие увеличения объема приводят лишь к малому росту мощности, сохраняется популярность меньших, более изящных языков, например Паскаля. Однако имеется насущная необходимость в эффективной методологии, которая помогла бы нам размышлять о программах, и ни один из тради-

ционных языков даже не подступился к удовлетворению этой потребности. В действительности традиционные языки создают ненужные помехи для наших суждений о программах.

В течение двадцати лет языки программирования неизменно развивались в одном и том же направлении, пока не дошли до нынешнего состояния «ожирения»; в результате изучение и изобретение языков программирования в значительной степени потеряли свою привлекательность. Напротив, теперь это излюбленная область деятельности для тех, кто предпочитает возиться с пухлыми перечнями подробностей вместо того, чтобы бороться за новые идеи. Дискуссии о языках программирования часто напоминают средневековые диспуты о числе ангелов, которые могут разместиться на кончике иглы, а не волнующие споры о фундаментально различных понятиях.

Многие творчески одаренные исследователи переключились с изобретения языков на изобретение средств их описания. К сожалению, им приходилось применять свои изящные новые средства преимущественно к изучению бородавок и родинок существующих языков. Достойно удивления, почему столь многие из нас, изучив отвратительные структуры типов традиционных языков с помощью изящного инструментария, разработанного Д. Скоттом, пассивно сохраняют верность этим структурам вместо того, чтобы энергично искать новые структуры.

Данная статья преследует две цели: во-первых, показать, что фундаментальные недостатки традиционных языков делают их выразительную слабость и их злокачественное разрастание неизбежными, и, во-вторых, предложить альтернативные пути исследований в направлении проектирования новых видов языков.

2. МОДЕЛИ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Основой всякого языка программирования является модель вычислительной системы, которой управляют программы на этом языке. Некоторые модели являются чистейшими абстракциями, некоторые представляются аппаратным обеспечением, а другие — компилирующими или интерпретирующими программами. Прежде чем вплотную заняться изучением традиционных языков, полезно дать краткий обзор существующих моделей как введение в текущее разнообразие альтернатив. Грубая классификация существующих моделей может основываться на намеченных ниже критериях.

2.1. КРИТЕРИИ ОЦЕНКИ МОДЕЛЕЙ

2.1.1. Основания. Существует ли изящное и точное математическое описание модели? Годится ли оно для доказательства полезных фактов относительно поведения модели? Или же модель настолько сложна, что ее описание является громоздким, и математик мало что может с ним сделать?

2.1.2. Историческая чувствительность. Включает ли модель понятие памяти, благодаря которому одна программа может запасать информацию, которая может повлиять на поведение более поздней программы? Иначе говоря, является ли модель исторически чувствительной?

2.1.3. Тип семантики. Осуществляет ли программа последовательные преобразования состояний (не являющихся программами) до тех пор, пока не будет достигнуто завершающее состояние (семантика смены состояний)? Являются ли состояния простыми или сложными? Может ли «программа» последовательно сводиться к более простым «программам», чтобы в конечном итоге превратиться в «программу в нормальной форме», которая представляет собой результат последовательного сведения (редукционная семантика)?

2.1.4. Ясность и концептуальная полезность программ. Являются ли программы модели ясными выражениями процесса или вычисления? Воплощаются ли в них концепции, помогающие нам формулировать утверждения и рассуждения о процессах?

2.2. КЛАССИФИКАЦИЯ МОДЕЛЕЙ

С помощью перечисленных критериев мы можем грубо охарактеризовать три класса моделей для вычислительных систем: простые операционные модели, аппликативные модели и модели фон Неймана.

2.2.1. Простые операционные модели. Примеры: машины Тьюринга, различные автоматы. *Основания*: точные и полезные. *Историческая чувствительность:* обладают памятью и исторической чувствительностью. *Семантика*: смена состояний с очень простыми состояниями. *Ясность программ*: программы неясные и концептуально бесполезные.

2.2.2. Аппликативные модели. Примеры: лямбда-исчисление Чёрча [5], система комбинаторов Карри [6], чистый Лисп [17], системы функционального программирования, описываемые в этой статье. *Основания*: точные и полезные. *Историческая чувствительность*: нет памяти, нет исторической чувствительности. *Семантика*: редукционная семантика, нет состояний. *Ясность*

программ: программы могут быть ясными и функционально полезными.

2.2.3. Модели фон Неймана. Примеры: компьютеры фон Неймана, традиционные языки программирования. *Основания*: сложные, громоздкие, бесполезные. *Историческая чувствительность*: обладают памятью, исторически чувствительны. *Семантика*: смена состояний со сложными состояниями. *Ясность программ*: программы могут быть умеренно ясными и не очень полезными концептуально.

Приведенная выше классификация, возможно, является грубой и спорной. Некоторые недавно появившиеся модели не удается без труда подогнать к какой-либо из этих категорий. Например, языки потоков данных, разработанные Эрвидном и Гостлоу [1], Деннисом [7], Косински [13] и другими исследователями, отчасти обладают свойствами, позволяющими отнести их к классу простых операционных моделей, но их программы яснее, чем в прежних моделях из этого класса, и, по-видимому, можно утверждать, что некоторые из них обладают редукционными семантиками. Во всяком случае, данная классификация будет служить грубой картой изучаемой территории. Мы сосредоточим внимание на аппликативных моделях и моделях фон Неймана.

3. КОМПЬЮТЕРЫ ФОН НЕЙМАНА

Для того чтобы понимать проблематику традиционных языков программирования, нам нужно сначала исследовать их интеллектуального прародителя, компьютер фон Неймана. Что такое компьютер фон Неймана? Когда фон Нейман и другие задумывали его более тридцати лет назад, это была изящная, практичная и объединяющая идея, которая упрощала ряд существовавших тогда инженерных и программистских задач. Хотя условия, породившие архитектуру этого компьютера, с тех пор радикально изменились, тем не менее мы по-прежнему идентифицируем понятие «компьютера» с этой концепцией тридцатилетней давности.

В своей простейшей форме компьютер фон Неймана состоит из трех частей: центрального процессорного устройства (ЦПУ), памяти и соединительной шины, которая может за один шаг передавать только одно слово между ЦПУ и памятью (и посылать некий адрес в память). Я предлагаю называть эту шину «*бутылочным горлышком*» (узким местом) *фон Неймана*. Задача программы состоит в том, чтобы неким существенным образом изменить содержимое памяти; если считать, что эта задача должна быть выполнена исключительно перекачивани-

ем одиночных слов туда и обратно через узость фон Неймана, то становится ясной причина такого названия.

Ирония ситуации состоит в том, что бóльшую часть потока через эту узость составляют не полезные данные, а всего лишь имена данных, а также операции и данные, служащие лишь для вычисления таких имен. Прежде чем слово можно будет послать через шину, его адрес должен находиться в ЦПУ; поэтому он должен либо быть послан через шину из памяти, либо генерироваться посредством некоторой операции ЦПУ. Если адрес посылается из памяти, то *адрес этого адреса* должен либо быть послан из памяти, либо генерироваться в ЦПУ, и т. д. С другой стороны, если адрес генерируется в ЦПУ, он должен генерироваться либо по фиксированному правилу (например, «добавить 1 к счетчику программы»), либо по команде, которая была послана через шину, в последнем случае ее адрес нужно было прежде послать... и т. д.

Разумеется, должен существовать менее примитивный способ внесения в память больших изменений, чем мельтешение множества слов туда и обратно через узость фон Неймана. Эта ши́на является не только узким местом для потока данных задачи, но, что более важно, и интеллектуальной узостью, которая привязывает нас к мышлению «слово за словом» вместо того, чтобы вдохновлять нас на мышление более крупными концептуальными частями решаемой задачи. Итак, программирование в основном сводится к планированию и спецификации огромного потока слов через узость фон Неймана, причем большая часть этого потока состоит не из самих значащих данных, а из сведений о том, где их искать.

4. ЯЗЫКИ ФОН НЕЙМАНА

Обычные языки программирования в основном являются высокоуровневыми, сложными версиями компьютера фон Неймана. Наша вера тридцатилетней давности, что существует только один вид компьютера, стала основой нашей уверенности в том, что существует только один вид языка программирования — традиционный язык фон Неймана. При всей существенности различия между Фортраном и Алголом-58 оно имеет меньше значения, чем тот факт, что оба этих языка основываются на программистском стиле компьютера фон Неймана. Хотя я упоминаю традиционные языки как «языки фон Неймана», чтобы отметить их первоисточник и стиль, я, конечно, не упрекаю великого математика за их сложность. Действительно, многие могли бы сказать, что я сам до некоторой степени несу ответственность за эту проблему.

В языках программирования фон Неймана переменные используются для имитации ячеек памяти компьютера; операторы управления выражают его команды передачи управления и проверки, а операторы присваивания имитируют вызов содержимого ячейки, запоминание и арифметику. Оператор присваивания представляет собой узость фон Неймана для языков программирования и вынуждает нас думать в терминах «слово за словом» примерно таким же образом, как на наше мышление воздействует существование в компьютере шины обмена данными между памятью и ЦПУ.

Рассмотрим типичную программу. Ее основу составляет ряд операторов присваивания, содержащих некоторые переменные с индексами. Каждый оператор присваивания порождает результат, состоящий из одного слова. Программа должна организовывать многократное выполнение этих операторов со сменой значений индексов, чтобы произвести желаемое итоговое изменение в памяти, поскольку она связана необходимостью изменять каждый раз только одно слово. Таким образом, программист имеет дело с потоком слов через узость присваиваний в соответствии с тем, как он проектирует вложенность управляющих операторов для обеспечения необходимых повторений.

Кроме того, оператор присваивания расщепляет программирование на два мира. Первый мир включает в себя правые части операторов присваивания. Это упорядоченный мир выражений, мир с полезными алгебраическими свойствами (если не учитывать того, что эти свойства часто нарушаются побочными эффектами). Это тот мир, в котором происходит большинство полезных вычислений.

Второй мир традиционных языков программирования — это мир операторов. Первичным оператором в этом мире является сам оператор присваивания. Все остальные операторы языка существуют для того, чтобы создать возможность выполнения вычисления, которое должно основываться на этой примитивной конструкции: на операторе присваивания.

Этот мир операторов неупорядочен, и у него мало полезных математических свойств. Структурное программирование можно считать скромной попыткой внести некий порядок в этот хаотический мир, но оно в малой степени способствует разрешению тех фундаментальных проблем, которые вносятся «пословным» стилем программирования фон Неймана с его примитивным использованием циклов, индексов и разветвлений потока управления.

Наша фиксация на языках фон Неймана сохранила преобладание компьютера фон Неймана, а наша зависимость от *него* сделала не-фон-неймановские языки неэкономичными и ограничила их развитие. Отсутствие законченных, эффективных стилей

программирования, опирающихся на принципы, отличные от принципов фон Неймана, обезоружило проектировщиков интеллектуальных основ новых компьютерных архитектур. (Эта проблема кратко обсуждается в разд. 15.)

Аппликативные вычислительные системы не стали основой проектирования компьютеров главным образом из-за отсутствия в них памяти и исторической чувствительности. Кроме того, в большинстве аппликативных систем в качестве основной операции используется операция подстановки лямбда-исчисления. Эта операция обладает фактически неограниченной выразительной мощью, но ее полная и эффективная реализация весьма затруднительна для проектировщиков компьютеров. К тому же при попытках оснастить аппликативные системы памятью и повысить их эффективность на компьютерах фон Неймана возникла тенденция к погружению этих систем в большие системы фон Неймана. Например, чистый Лисп часто становится подмножеством больших расширений, обладающих многими свойствами систем фон Неймана. Получающиеся в результате сложные системы мало что могут подсказать разработчику компьютеров.

5. СРАВНЕНИЕ ПРОГРАММ ФОН НЕЙМАНА С ФУНКЦИОНАЛЬНЫМИ ПРОГРАММАМИ

Чтобы получить более детальное представление о некоторых недостатках языков фон Неймана, сравним обычную программу вычисления внутреннего произведения с соответствующей функциональной программой, написанной на простом языке, который будет детализирован впоследствии.

5.1. ПРОГРАММА ФОН НЕЙМАНА ДЛЯ ВНУТРЕННЕГО ПРОИЗВЕДЕНИЯ

$c := 0$
for i : 1 **step** 1 **until** n **do**
$c := c + a[i] \times b[i]$

Заслуживают упоминания следующие свойства этой программы.

(а) Ее операторы действуют на скрытые «состояния» в соответствии со сложными правилами.

(б) Она не является иерархической. За исключением правой части оператора присваивания, она не конструирует сложных объектов из более простых. (Впрочем, большие программы часто делают это.)

(в) Она динамическая и итеративная. Человек должен мысленно исполнить ее, чтобы понять ее работу.

(г) Она проводит вычисления пословно, повторяя (присваивание) и модифицируя (переменную i).

(д) Часть данных (число n) содержится в программе; поэтому ей недостает общности и она работает только с векторами длины n.

(е) Она именует свои аргументы; ее можно использовать только для векторов a и b. Чтобы она стала общей, требуется описание процедуры. При этом возникают сложные проблемы (например, вызов по имени вместо вызова по значению).

(ж) Ее операции «внутреннего хозяйства» представляются символами, помещенными в разных местах (в операторе **for** и в индексах оператора присваивания). Из-за этого невозможно сосредоточить операции внутреннего хозяйства, которые являются наиболее общими из всех операций, в единых, мощных, широко используемых операциях. Таким образом, при программировании таких операций всегда приходится начинать снова «в лоб», выписывая «**for** i: =...» и «**for** j: =...», а затем операторы присваивания, пестрящие индексами i и j.

5.2. ФУНКЦИОНАЛЬНАЯ ПРОГРАММА ДЛЯ ВНУТРЕННЕГО ПРОИЗВЕДЕНИЯ

Def Внутреннее произведение = (Вставить +) ∘ (Применить ко всем $\times$) ∘ Транспозиция или в сокращенной форме

Def IP = (/+) ∘ ($\alpha\times$) ∘ Trans.

Композиция (∘), Вставить (/) и Применить ко всем (α) являются *функциональными формами*, которые комбинируют существующие функции для образования новых. Так, $f \circ g$ — это функция, получаемая применением сначала g, затем f, а αf — функция, получаемая применением f к каждому *слагаемому* аргумента. Если мы пишем $f{:}x$ для обозначения результата применения f к объекту x, то можем объяснить каждый шаг вычисления внутреннего произведения IP применительно к паре векторов ⟪1, 2, 3⟩, ⟨6, 5, 4⟫ следующим образом:

IP:⟪1, 2, 3⟩, ⟨6, 5, 4⟫ =
Описание IP ⇒ (/+) ∘ ($\alpha\times$) ∘ Trans:⟪1, 2, 3⟩, ⟨6, 5, 4⟫
Результат композиции, ∘ ⇒ (/+):(($\alpha\times$): Trans:⟪1, 2, 3⟩⟨6, 5, 4⟫))
Применение транспозиции ⇒ (/+):(($\alpha\times$):⟪1, 6⟩, ⟨2, 5⟩, ⟨3, 4⟫)
Результат Применить ко всем, α ⇒ (/+):⟨×:⟨1, 6⟩, ×:⟨2, 5⟩, ×:⟨3, 4⟩⟩
Применение $\times$ ⇒ (/+):⟨6, 10, 12⟩

Результат Вставить, / $\Rightarrow +:\langle 6,+:\langle 10,12\rangle\rangle$
Применение + $\Rightarrow +:\langle 6,22\rangle$
Применение + снова $\Rightarrow 28$

Сравним свойства этой программы со свойствами программы фон Неймана.

(а) Она оперирует только своими аргументами. Здесь нет скрытых состояний или сложных правил перехода. Имеются только два вида правил, одно для применения функции к ее аргументу, а другое для получения функции, обозначаемой такой функциональной формой, как композиция $f\circ g$ или Применить ко всем, αf, если известны параметры форм, т. е. функции f и g.

(б) Она является иерархической, поскольку строится из трех более простых функций (+, $\times$, Trans) и трех функциональных форм $f\circ g$, αf и $/f$.

(в) Она является статической и неитеративной в том смысле, что ее структура удобна для ее понимания без мысленного исполнения. Например, если некто понимает действие форм $f\circ g$ и αf и функций $\times$ и Trans, то он понимает действие $\alpha\times$ и $(\alpha\times)\circ$Trans и т. д.

(г) Она оперирует целыми концептуальными блоками, а не словами; она состоит из трех этапов, ни один этап не повторяется.

(д) Она не включает данных; она является вполне общей; она работает для любой пары векторов одинаковой размерности.

(е) Она не именует свои аргументы; она применима к любой паре векторов без всяких описаний процедур или сложных правил подстановок.

(ж) Она использует формы и функции внутреннего хозяйства, которые универсально полезны во многих других программах; на самом деле только операции + и $\times$ не имеют отношения к внутреннему хозяйству. Эти формы и функции могут комбинироваться с другими, создавая операции внутреннего хозяйства более высокого уровня.

Раздел 14 содержит набросок некой системы, спроектированной для того, чтобы сделать такой функциональный стиль программирования доступным в простой по структуре, исторически чувствительной системе, но потребуется еще много труда, прежде чем этот функциональный стиль сможет стать основой изящных и практичных языков программирования. В настоящее время проведенные выше сравнения выявляют ряд серьезных недостатков языков фон Неймана и могут служить отправной точкой для усилий по преодолению их нынешней тучности и вялости.

6. СТРУКТУРЫ ЯЗЫКОВ В СРАВНЕНИИ С ИЗМЕНЯЕМЫМИ ЧАСТЯМИ

Будем различать две части языка программирования. Первой является *структура*, которая задает общие правила системы, а вторая — это *изменяемая часть*, существование которой подразумевается структурой, но конкретное поведение которой не специфицируется. Например, оператор **for** и почти все остальные операторы являются частью структуры Алгола, а библиотечные функции и описываемые пользователем процедуры представляют собой изменяемую часть. Итак, структура языка описывает его фиксированные свойства и общую среду для его изменяемых свойств.

Теперь предположим, что язык обладает небольшой структурой, которая может объединять много разнообразных мощных средств целиком в качестве изменяемых частей. Тогда такая структура могла бы поддерживать много разных выразительных средств и стилей, не изменяясь при этом сама. В отличие от этой приятной возможности языки фон Неймана, по-видимому, всегда обладают огромной структурой и весьма ограниченными изменяемыми частями. Почему так получается? Ответ на этот вопрос затрагивает проблемы, связанные с языками фон Неймана.

Первая проблема возникает из-за стиля фон Неймана пословного программирования, который требует, чтобы слова сновали взад и вперед к состоянию и от него, совсем как в потоке через узость фон Неймана. Таким образом, язык фон Неймана должен обладать семантикой, тесно увязанной с состоянием, причем всякая подробность вычислений изменяет состояние. Вследствие такой тесной связи семантики с состояниями всякая деталь каждого свойства должна быть встроена в состояние и в его правила перехода.

Итак, каждое свойство языка фон Неймана должно быть закодировано с ошеломляющими подробностями в структуре языка. Более того, многие сложные свойства нужно подпирать слабым фундаментом пословного стиля. Результатом является неизбежно жесткая и огромная структура языка фон Неймана.

7. ИЗМЕНЯЕМЫЕ ЧАСТИ И КОМБИНАЦИОННЫЕ ФОРМЫ

Вторая проблема, связанная с языками фон Неймана, состоит в слишком малой выразительной силе их изменяемых частей. Красноречивым доказательством этого являются их раблезианские размеры. К тому же, если бы проектировщик знал, что все усложненные свойства, которые он теперь встраивает в структуру, могли быть впоследствии добавлены в качестве

изменяемой части, он не стал бы столь усердно втискивать их в структуру.

Возможно, самым важным аспектом обеспечения мощи изменяемой части языка является доступность комбинационных форм, которые в общем случае могут служить для построения новых процедур из старых. Языки фон Неймана обеспечивают только примитивные комбинационные формы, а структура фон Неймана создает препятствия для их полноценного использования.

Одним из препятствий применению комбинационных форм является водораздел между миром выражений и миром операторов в языках фон Неймана. Естественно, функциональные формы принадлежат к миру выражений, но вне зависимости от их мощи они могут строить только такие выражения, которые порождают однословный результат. И именно в мире операторов эти однословные результаты должны сочетаться в итоговом результате. Сочетание отдельных слов — это не то, о чем нам на самом деле следовало бы думать, тем не менее это большая часть программирования любой задачи на языках фон Неймана. Чтобы помочь сборке общего результата из отдельных слов, эти языки обеспечивают некоторые примитивные комбинационные формы в мире операторов — операторы **for**, **while** и **if — then — else**, но водораздел между двумя мирами мешает тому, чтобы комбинационные формы в каждом из миров обрели полную силу, которую они могли бы получить в неразделенном мире.

Второе препятствие применению комбинационных форм в языках фон Неймана состоит в использовании в них хитроумных соглашений именования, которые дополнительно усложняются правилами подстановок, требуемых в процедурах вызова. Для каждой из них нужно, чтобы в структуру языка был встроен сложный механизм, обеспечивающий возможность правильной интерпретации простых переменных, переменных с индексами, указателей, имен файлов, имен процедур, формальных параметров, вызываемых по значению, формальных параметров, вызываемых по имени, и т. д. Все эти имена, соглашения и правила затрудняют использование простых комбинационных форм.

8. ЯЗЫК APL В СРАВНЕНИИ С ПОСЛОВНЫМ ПРОГРАММИРОВАНИЕМ

Сказав так много о пословном программировании, я должен теперь что-нибудь сказать о языке APL [12]. Мы очень обязаны К. Айверсону, который показал нам, что существуют программы, не являющиеся пословными и не зависящие от лямб-

да-выражений, и познакомил нас с применением новых функциональных форм. А поскольку операторы присваивания языка APL могут запоминать массивы, то воздействие их функциональных форм выходит за рамки одиночного присваивания.

Однако, к сожалению, язык APL все же расщепляет программирование на мир выражений и мир операторов. Поэтому старание писать однострочные программы частично обусловлено желанием оставаться в более упорядоченном мире выражений. В языке APL имеются ровно три функциональные формы, называемые внутренним произведением, внешним произведением и сведением. Иногда их трудно использовать, их количество недостаточно, и их употребление ограничивается миром выражений.

Наконец, семантика языка APL по-прежнему тесно связана с состоянием. Поэтому, несмотря на большую простоту и мощь языка, его структура характеризуется сложностью и жесткостью языков фон Неймана.

9. НЕДОСТАТОЧНОСТЬ ПОЛЕЗНЫХ МАТЕМАТИЧЕСКИХ СВОЙСТВ В ЯЗЫКАХ ФОН НЕЙМАНА

До сих пор мы обсуждали большой размер и негибкость языков фон Неймана; другим их существенным неудобством является недостаточность математических свойств и вызываемая ими затруднительность рассуждений о программах. Несмотря на изобилие отменных публикаций с доказательствами фактов о программах, языки фон Неймана почти полностью лишены полезных в этом отношении свойств и обладают множеством свойств, являющихся помехами (например, побочные эффекты, различные имена для одних и тех же объектов).

Денотационная семантика [23] и ее обоснование [20, 21] обеспечивают исключительно полезное понимание скрытых в программах областей определения и функциональных пространств. При применении к «практичному» языку (например, к языку «рекурсивных программ» из [16]) ее основания дают эффективный инструментарий для описания языка и доказательства свойств программ. С другой стороны, при применении к языку фон Неймана она предоставляет точное семантическое описание и полезна для обнаружения слабых мест в языке. Однако сложность языка отражается в сложности описания, которое представляет собой ошеломляющее нагромождение правил подстановок, областей определения, функций и уравнений, лишь немногим более удобных для доказательства фактов о программах, чем стандартное руководство по языку, поскольку в таком руководстве все-таки меньше двусмысленностей.

Аксиоматическая семантика [11] в точности воспроизводит неизящные свойства программ фон Неймана (т. е. преобразования на состояниях) в виде преобразований на предикатах. Тем самым изменяется не сама пословная итеративная игра, а только поле для игры. Сложность этой аксиоматической игры в доказательстве фактов о программах фон Неймана делает успехи ее участников еще более восхитительными. Кроме изобретательности, их успех основывается на двух дополнительных факторах. Во-первых, игра ограничивается малыми, слабыми подмножествами языков фон Неймана со значительно более простыми состояниями, чем в реальных языках. Во-вторых, новое поле для игры (предикаты и их преобразования) более богато, упорядоченно и эффективно, чем прежнее (состояния и их преобразования). Но ограничив богатство этой игры и перенеся ее на более эффективную область, мы утрачиваем возможность работать с реальными программами (с неизбежными сложностями вызовов процедур и несовпадений имен) и к тому же не исключим неуклюжие свойства основного стиля фон Неймана. По мере того как аксиоматическая семантика обобщается, чтобы покрывать более обширные подмножества типичного языка фон Неймана, она начинает терять свою эффективность из-за возрастания требуемой сложности.

Итак, денотационная и аксиоматическая семантики представляют собой описательные формализмы, опирающиеся на изящные и мощные понятия, но использование их для описания языка фон Неймана не может породить изящного и мощного языка, подобного тому, как употребление изящных и мощных машин для построения игрушечных автомобильчиков не может породить изящный и мощный современный автомобиль.

В любом случае для доказательства фактов о программах используется язык логики, а не язык программирования. Доказательства ведут речь *о* программах, но не могут использовать их непосредственно, потому что аксиомы языков фон Неймана являются столь неконструктивными. Напротив, многие обычные доказательства выводятся алгебраическими методами. Для этих методов требуется язык, обладающий определенными алгебраическими свойствами. Потом алгебраические законы могут применяться довольно механически для преобразования проблемы в ее решение. Например, для решения уравнения

$$ax+bx=a+b$$

относительно x (при условии, что $a+b\neq 0$) мы механически применяем последовательно законы дистрибутивности, тождества и сокращения, чтобы получить

$$(a+b)x=a+b$$

$$(a+b)x=(a+b)1$$
$$x=1.$$

Итак, мы доказали, что $x=1$, не отходя от «языка» алгебры. Языки фон Неймана со своим нелепым синтаксисом представляют мало таких возможностей для преобразования программ.

Как мы увидим позднее, программы могут быть выражены на языке, с которым связана некоторая алгебра. Эта алгебра может служить для преобразования программ и для решения некоторых уравнений, в которых «неизвестными» являются программы, примерно таким же образом, как решаются уравнения в высшей алгебре. В алгебраических преобразованиях и доказательствах используется язык самих программ, а не язык логики для изложения фактов о программах.

10. КАКОВЫ АЛЬТЕРНАТИВЫ ДЛЯ ЯЗЫКОВ ФОН НЕЙМАНА?

Прежде чем переходить к рассмотрению альтернатив языков фон Неймана, замечу, что меня огорчает необходимость проведенного выше негативного и не слишком точного обсуждения этих языков. Но благодушное принятие большинством из нас этих громоздких, слабых языков слишком долго озадачивало и расстраивало меня. Я расстраиваюсь из-за того, что это принятие поглотило огромные усилия, направленные на то, чтобы сделать такие языки еще жирнее, хотя эти усилия лучше было бы направить на поиск новых структур. Поэтому я попытался проанализировать некоторые из основных недостатков традиционных языков и показать, что от этих недостатков нельзя избавиться, если мы не найдем новый вид структуры языка.

В поиске альтернативы для традиционных языков нам нужно осознать, что система не может быть исторически чувствительной (допускать влияние выполнения одной программы на поведение следующей программы), если система не обладает некоторым состоянием (которое первая программа может изменять, а вторая может воспринимать). Итак, исторически чувствительная модель вычислительной системы должна обладать семантикой смены состояний хотя бы в этом слабом смысле. Но из этого не следует, что всякое вычисление должно существенно зависеть от сложного состояния, причем для каждой малой части вычислений требуется много изменений состояний (как в языках фон Неймана).

Чтобы проиллюстрировать некоторые альтернативы языкам фон Неймана, я предлагаю обозреть класс исторически чувствительных вычислительных систем, причем каждая система (а) характеризуется слабой зависимостью от семантики смены состояний, т. е. изменение состояния производится только один

раз в течение большого вычисления; (б) обладает просто структурированными состояниями и простыми правилами перехода; (в) существенно зависит от лежащей в ее основе аппликативной системы как в обеспечении базового языка программирования системы, так и в описании смены ее состояний.

Эти системы, которые я называю аппликативными системами переходов состояний (АСПС), описаны в разд. 14. Такие простые системы свободны от многих сложностей и слабостей языка фон Неймана и предусматривают мощный и обширный набор изменяемых частей. Однако они упоминаются только как грубые примеры из обширной области не-фон-неймановских систем с разнообразными притягательными свойствами. Я изучал эту область в течение последних трех или четырех лет и еще не нашел удовлетворительного решения для многих противоречивых требований, которым должен удовлетворять хороший язык. Однако я полагаю, что этот поиск указал полезный подход к проектированию не-фон-неймановских языков.

Данный подход включает четыре элемента, которые могут быть подытожены следующим образом.

(а) *Функциональный стиль программирования без переменных.* Описывается простая неформальная система функционального программирования (ФП). Она основывается на использовании комбинационных форм программ ФП. Приводится несколько программ для иллюстрации функционального программирования.

(б) *Алгебра функциональных программ.* Описывается алгебра, в которой переменные обозначают функциональные программы, а операции являются функциональными формами ФП, т. е. комбинационными формами программ ФП. Формулируются некоторые законы этой алгебры. Приводятся теоремы и примеры, которые показывают, как конкретные функциональные выражения могут преобразовываться в эквивалентные бесконечные разложения, объясняющие поведение функции. Алгебра ФП сравнивается с алгебрами, соответствующими классическим функциональным системам Чёрча и Карри.

(в) *Формальная система функционального программирования.* Описывается формальная система (ФФП), обобщающая возможности упомянутых выше неформальных систем ФП. Таким образом, система ФФП является точно определенной системой, обеспечивающей возможность использовать стиль функционального программирования систем ФП и их алгебру программ. Системы ФФП могут служить основой для аппликативных систем переходов состояний.

(г) *Аппликативные системы переходов состояний.* См. выше.

В оставшейся части статьи описываются эти четыре элемента, и в заключение приводится резюме работы.

11. СИСТЕМЫ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ (СИСТЕМЫ ФП)

11.1. ВВЕДЕНИЕ

В этом разделе мы даем неформальное описание одного класса простых аппликативных систем программирования, называемых функциональными системами (ФП), в которых «программами» являются просто функции без переменных. Описание сопровождается некоторыми примерами и обсуждением различных свойств систем ФП.

В системе ФП находит применение фиксированное множество комбинационных форм, называемых функциональными формами. Они в сочетании с простыми описаниями являются единственными средствами построения новых функций из уже существующих. В них не используются переменные или правила подстановок, и они становятся операциями соответствующей алгебры программ. Все функции системы ФП относятся к одному типу: они отображают объекты на объекты и всегда работают с одним аргументом.

Напротив, в системе, основанной на лямбда-исчислении, с целью построения новых функций применяется лямбда-выражение с соответствующим множеством правил подстановок для переменных. Лямбда-выражение (с его правилами подстановок) в состоянии описать все возможные вычислимые функции всех мыслимых типов и с любым количеством аргументов. Эта свобода и выразительность чревата недостатками наряду с очевидными преимуществами. Она аналогична мощи неограниченных операторов управления в традиционных языках: вместе с неограниченной свободой приходит хаос. Если некто постоянно изобретает новые комбинационные формы применительно к возникающим ситуациям, как можно поступать в лямбда-исчислении, то он никогда не познакомится со стилем или полезными свойствами немногих комбинационных форм, подходящих ко всем случаям. Подобно тому как структурное программирование избегает многих управляющих операторов для получения программ с более простой структурой, лучшими свойствами и единообразными методами понимания их поведения, так и функциональное программирование избегает лямбда-выражений, подстановок и множественных типов функций. Благодаря этому получаются программы, построенные из хорошо знакомых функциональных форм с известными полезными свойствами. Эти программы структурированы таким образом, что часто можно понять и доказать их поведение механическим использованием алгебраических приемов, подобных тем, которые применяются при решении задач высшей алгебры.

В отличие от большинства конструкций программирования функциональные формы не нужно выбирать применительно к тому или иному случаю. Поскольку они являются операциями соответствующей алгебры, мы выбираем лишь такие функциональные формы, которые не только обеспечивают выразительные конструкции программирования, но и обладают привлекательными алгебраическими свойствами: они выбираются для максимизации силы и полезности алгебраических законов, связывающих их с другими функциональными формами системы.

В приводимом ниже описании мы не станем пунктуально различать (а) функциональный символ или выражение и (б) обозначаемую им функцию. Символы и выражения, служащие для обозначения функций, будут указываться посредством их использования. В разд. 13 описывается формальное обобщение систем ФП (системы ФФП); они могут служить для разъяснения любых неопределенностей относительно систем ФП.

11.2. ОПИСАНИЕ

Система ФП включает следующее:

(1) множество *O объектов*;

(2) множество F *функций* f, которые отображают объекты на объекты;

(3) операцию, *применение*;

(4) множество **F** *функциональных форм*; они служат для комбинирования существующих функций или объектов для формирования новых функций из **F**;

(5) множество *D описаний*, определяющих некоторые функции в **F** и присваивающих каждой из них имя.

Далее следует неформальное описание с примерами всех этих составных частей.

11.2.1. Объекты, O. *Объект x* является либо *атомом*, либо *последовательностью* $(x_1, \ldots, x_n)$, элементы которой представляют собой объекты, либо $\perp$ («основой», или «неопределенностью»). Таким образом, выбор множества A атомов определяет множество объектов. Мы будем считать, что A является множеством непустых строк из прописных букв, цифр и специальных символов, не используемых в нотации системы ФП. Некоторые из этих строк принадлежат к классу атомов, называемых «числами». Атом $\varnothing$ служит для обозначения пустой последовательности и является единственным объектом, который представляет собой одновременно и атом, и последовательность. Атомы *T* и *F* служат для обозначения «истины» и «лжи».

На конструирование объектов накладывается одно важное ограничение: если x — это последовательность с элементом $\perp$,

то $x = \bot$. Иначе говоря, «конструктор последовательностей» является «сохраняющим $\bot$». Итак, ни одна последовательность в собственном смысле не содержит $\bot$ в качестве элемента.

Примеры объектов

$\bot$ 1.5 $\varnothing$ *AB*3 $\langle AB, 1, 2, 3\rangle\langle A,\langle B\rangle,C\rangle,D\rangle$ $\langle A, \bot\rangle = \bot$

11.2.2. Применение. Система ФП включат единственную операцию, применение. Если f — это функция и x — объект, то $f:x$ представляет собой *применение* и обозначает объект, являющийся результатом применения f к x. Здесь f — *оператор* применения, а x — *операнд*.

Примеры применений

$+:\langle 1, 2\rangle = 3$ $\mathrm{tl}:\langle A, B, C\rangle = \langle B, C\rangle$ $1:\langle A, B, C\rangle = A$ $2:\langle A, B, C\rangle = B$

11.2.3. Функции, F. Все функции f из F отображают объекты на объекты и *«сохраняют основу»*: $f:\bot = \bot$ для всех f из F. Любая функция из F является или *примитивной*, т. е. поставляемой вместе с системой, или *описываемой* (см. ниже), или *функциональной формой* (см. ниже).

Иногда бывает полезно различать два случая, в которых $f:x = \bot$. Если вычисление для $f:x$ завершается и порождает объект $\bot$, мы говорим, что функция f *неопределена* в x, т. е. f завершается, но не дает содержательного значения в x. В противном случае мы говорим, что f *незавершима* в x.

Примеры примитивных функций. Наша цель состоит в том, чтобы обеспечить системы ФП широко применимыми и мощными примитивными функциями, а не слабыми примитивными функциями, которые затем могли бы быть использованы для описания полезных функций. В последующих примерах описываются некоторые типичные примитивные функции, многие из которых применяются в дальнейших примерах программ. В следующих описаниях мы употребляем вариант условного выражения Маккарти [17]; итак, мы пишем:

$p_1 \to e_1; \ldots; p_n \to e_n; e_{n+1}$.

вместо выражения Маккарти

$(p_1 \to e_1, \ldots, p_n \to e_n, T \to e_{n+1})$.

Следующие описания должны иметь место для всех объектов x, x_i, y, y_i, z, z_i.

Функции селектора

$1:x \equiv x = \langle x_1, \ldots, x_n\rangle \to x_1; \bot$

и для любого положительного целого значения s

$s : x \equiv x = \langle x_1, \ldots, x_n \rangle \;\&\; n \geqslant s \rightarrow x_s;\ \bot$

Таким образом, например, $3{:}\langle A, B, C\rangle = C$ и $2{:}\langle A\rangle = \bot$. Заметим, что символы функций 1, 2 и т. д. отличаются от атомов *1*, *2* и т. д.

Хвост (*tail*)

$tl : x \equiv x = \langle x_1 \rangle \rightarrow \varnothing;\ x = \langle x_1, \ldots, x_n \rangle \;\&\; n \geqslant 2 \rightarrow \langle x_2, \ldots, x_n \rangle;\ \bot$

Тождество (*identity*)

$id : x \equiv x$

Атом (*atom*)

atom : $x \equiv x$ является атомом $\rightarrow T;\ x \neq \bot \rightarrow F;\ \bot$

Равняется (*equals*)

$eq : x \equiv x = \langle y, z \rangle \;\&\; y = z = z \rightarrow T;\ x = \langle y, z \rangle \;\&\; y \neq z \rightarrow F;\ \bot$

Нуль (*null*)

$null : x \equiv x = \varnothing \rightarrow T;\ x \neq \bot \rightarrow F;\ \bot$

Обращение (*reverse*)

$reverse : x \equiv x = \varnothing \rightarrow \varnothing;\ x = \langle x_1, \ldots, x_n \rangle \rightarrow \langle x_n, \ldots, x_1 \rangle;\ \bot$

Дистрибутивно слева, дистрибутивно справа (*distl, distr*)

$distl : x \equiv x = \langle y, \varnothing \rangle \rightarrow \varnothing;\ x = \langle y, (z_1, \ldots, z_n \gg \rightarrow \ll y, z_1 \rangle, \ldots$
$\ldots, \langle y, z_n \gg;\ \bot$

$distr : x \equiv x = \langle \varnothing, y \rangle \rightarrow \varnothing;\ x = \ll y_1, \ldots, y_n \rangle, z \rangle \rightarrow \ll y_1, z \rangle, \ldots$
$\ldots, \langle y_n, z \gg;\ \bot$

Длина (*length*)

$length : x \equiv x = \langle x_1, \ldots, x_n \rangle \rightarrow n;\ x = \varnothing \rightarrow 0;\ \bot$

Прибавить, вычесть, умножить и разделить (*add, subtract, multiply and divide*)

$+ : x \equiv x = \langle y, z \rangle \;\&\; y, z$ являются числами $\rightarrow y + z;\ \bot$
$- : x \equiv x = \langle y, z \rangle \;\&\; y, z$ являются числами $\rightarrow y - z;\ \bot$
$\times : x \equiv x = \langle y, z \rangle \;\&\; y, z$ являются числами $\rightarrow y \times z;\ \bot$
$\div : x \equiv x = \langle y, z \rangle \;\&\; y, z$ являются числами $\rightarrow y \div z;\ \bot$ (где $y \div 0 = = \bot$)

Транспозиция (*transpose*)

$trans : x \equiv x = \langle \varnothing, \ldots, \varnothing \rangle \rightarrow \varnothing;\ x = \langle x_1, \ldots, x_n \rangle \rightarrow \langle y_1, \ldots, y_m \rangle;\ \bot$

где

$x_i = \langle x_{i1}, \ldots, x_{im} \rangle$ и $y_j = \langle x_{1j}, \ldots, x_{nj} \rangle$, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$

и, или, отрицание (*and, or, not*)

$\text{and} : x \equiv x = \langle T, T \rangle \to T;\ x = \langle T, F \rangle \vee x = \langle F, F \rangle \to F;\ \bot$

и т. д.

Присоединить слева, присоединить справа [*append left; append right*)

$\text{apndl} : x \equiv x = \langle y, \varnothing \rangle \to \langle y \rangle;\ x = \langle y, \langle z_1, \ldots, z_n \rangle \rangle \to \langle y, z_1, \ldots, z_n \rangle;\ \bot$
$\text{apndr} : x \equiv x = \langle \varnothing, z \rangle \to \langle z \rangle;\ x = \langle \langle y_1, \ldots, y_n \rangle, z \rangle \to \langle y_1, \ldots, y_n, z \rangle;\ \bot$

Правые селекторы, правый хвост (*right selectors; right tail*)

$1r : x \equiv x = \langle x_1, \ldots, x_n \rangle \to x_n;\ \bot$
$2r : x \equiv x = \langle x_1, \ldots, x_n \rangle\ \&\ n \geqslant 2 \to x_{n-1};\ \bot$

и т. д.

$\text{tlr} : x \equiv x = \langle x_1 \rangle \to \varnothing;\ x = \langle x_1, \ldots, x_n \rangle\ \&\ n \geqslant 2 \to \langle x_1, \ldots, x_{n-1} \rangle;\ \bot$

Поворот влево, поворот вправо (*rotate left; rotate right*)

$\text{rotl} : x \equiv x = \varnothing \to \varnothing;\ x = \langle x_1 \rangle \to \langle x_1 \rangle;$
$x = \langle x_1, \ldots, x_n \rangle\ \&\ n \geqslant 2 \to \langle x_2, \ldots, x_n, x_1 \rangle;\ \bot$

и т. д.

11.2.4. Функциональные формы, F. Функциональная форма представляет собой выражение, означающее функцию; такая функция зависит от функций или объектов, которые являются *параметрами* выражения. Так, например, если f и g — это любые функции, то $f \circ g$ представляет собой функциональную форму, *композицию* f и g; f и g являются ее параметрами, и она обозначает функцию, такую, что для любого объекта x

$(f \circ g) : x = f : (g : x).$

Для некоторых функциональных форм параметрами могут быть объекты. Например, для любого объекта x, $\bar{x}$ является функциональной формой, *константной* функцией от x, так что для любого объекта y

$\bar{x} : y \equiv y \bot \to \bot;\ x.$

В частности, $\overline{\bot}$ — это функция «всюду $\bot$».

Ниже мы приводим некоторые функциональные формы, многие из которых используются позднее в этой статье. Мы применяем символы p, f и g с индексами и без индексов для обозначения произвольных функций, а x, x_1, ..., x_n, y являются произвольными объектами. Квадратные скобки [...] служат

для указания функциональной формы для *конструкции*, которая обозначает функцию, тогда как угловые скобки $\langle \ldots \rangle$ обозначают последовательности, которые являются объектами. Круглые скобки применяются и в конкретных функциональных формах (например, в *условии*), и в общем случае для указания группирования.

Композиция

$$(f \circ g) : x \equiv g : (g : x)$$

Конструкция

$$[f_1, \ldots, f_n] : x \equiv \langle f_1 : x, \ldots, f_n : x \rangle$$

(Напомним, что поскольку $\langle \ldots, \bot, \ldots \rangle$ и все функции сохраняют $\bot$, то $[f_1, \ldots, f_n\rangle$ тоже обладает этим свойством.)

Условие

$$(p \rightarrow f;\ g) : x \equiv (p : x) = T \rightarrow f : x;\ (p : x) = F \rightarrow g : x;\ \bot$$

Условные *выражения* (используемые вне систем ФП для описания их функций) и условие *функциональной формы* идентифицируются знаком «→». Это совсем разные, хотя и тесно связанные понятия, как показано в приведенных выше определениях. Но не должно возникать никакой путаницы, потому что все элементы условного выражения обозначают значения, тогда как все элементы условия функциональной формы обозначают функции, но заведомо не значения. Если не возникает никакой двусмысленности, мы опускаем ассоциирующиеся справа скобки; например, мы пишем

$p_1 \rightarrow f_1;\ p_2 \rightarrow f_2;\ g$ вместо $(p_1 \rightarrow f_1;\ (p_2 \rightarrow f_2;\ g))$.

Константа (Здесь x — это объектный параметр.)

$$\bar{x} : y \equiv y = \bot \rightarrow \bot;\ x$$

Включение

$$/f : x \equiv x = \langle x_1 \rangle \rightarrow x_1;$$
$$x = \langle x_1, \ldots, x_n \rangle\ \&\ n \geqslant 2 \rightarrow f : \langle x_1,\ /f : \langle x_2, \ldots, x_n \rangle\rangle;\ \bot$$

Если f содержит единственный правый элемент $u_f \neq \bot$, где $f : \langle x, u_f \rangle \in \{x, \bot\}$ для всех объектов x, то приведенное выше определение обобщается: $/f : \varnothing = u_f$. Итак,

$$/+ ; \langle 4, 5, 6 \rangle = + : \langle 4, + : \langle 5, /+ : \langle 6 \rangle \rangle \rangle = + : \langle 4, + : \langle 5, 6 \rangle \rangle = 15$$
$$/+ : \varnothing = 0$$

Применить ко всем

$$\alpha f : x \equiv x = \varnothing \rightarrow \varnothing;\ x = \langle x_1, \ldots, x_n \rangle \rightarrow \langle f : x_1, \ldots, f : x_n \rangle;\ \bot$$

Двоичное в единичное (x — объектный параметр)

$(\mathrm{bu}\, f x) : y \equiv f : \langle x, y \rangle$

Таким образом,

$(\mathrm{bu}+1) : x = 1 + x$

Пока (*while*)

$(\mathrm{while}\, p\, f) : x \equiv p : x = T \rightarrow (\mathrm{while}\, p\, f) : (f : x);\ p : x = F \rightarrow x;\ \bot$

Представленные выше функциональные формы обеспечивают эффективный метод вычисления значений обозначаемых ими функций (если они завершаются) при условии, что имеется возможность эффективно применять их параметры.

11.2.5. Определения. *Определением* в системе ФП является выражение вида

Def $l = r$

где левая часть l представляет собой еще не использованный символ функции, а правая часть r является функциональной формой (которая может зависеть от l). Оно выражает тот факт, что символ l должен обозначать функцию, задаваемую посредством r. Так, определение **Def** lastl = 1∘reverse определяет функцию lastl, которая порождает последний элемент последовательности (или $\bot$). Аналогично

Def last = null∘tl→1; last∘tl

описывает функцию last, такую же, как lastl. Ниже подробно показано, как это описание можно было бы применить для вычисления $\mathrm{last} : \langle 1, 2 \rangle$:

$\mathrm{last} : \langle 1, 2 \rangle$	=
определение last	$\Rightarrow (\mathrm{null} \circ \mathrm{tl} \rightarrow 1;\ \mathrm{last} \circ \mathrm{tl}) : \langle 1, 2 \rangle$
действие формы $(p \rightarrow f;\ g)$	$\Rightarrow \mathrm{last} \circ \mathrm{tl} : \langle 1, 2 \rangle$
так как $\mathrm{null} \circ \mathrm{tl} : \langle 1, 2 \rangle = \mathrm{null} : \langle 2 \rangle = F$	
действие формы $f \circ g$	$\Rightarrow \mathrm{last} : (\mathrm{tl} : \langle 1, 2 \rangle)$
определение примитивной хвостовой функции	$\Rightarrow \mathrm{last} : \langle 2 \rangle$
определение last	$\Rightarrow (\mathrm{null} \circ \mathrm{tl} \rightarrow 1;\ \mathrm{last} \circ \mathrm{tl});\ \langle 2 \rangle$
действие вида $(p \rightarrow f;\ g)$	$\Rightarrow 1 : \langle 2 \rangle$
так как $\mathrm{null} \circ \mathrm{tl} : \langle 2 \rangle = \mathrm{null} : \varnothing = T$	
определение селектора 1	$\Rightarrow 2$

Выше проиллюстрировано простое правило: для применения описанного символа замените его правой частью его определения. Разумеется, некоторые определения могут относиться к незавершаемым функциям. Множество D определений *корректно*, если никакие две левые части не совпадают.

11.2.6. Семантика. Из сказанного выше можно видеть, что система ФП определяется выбором следующих множеств: (а) Множество атомов A (которым определяется множество объектов). (б) Множество примитивных функций P. (в) Множество функциональных форм F. (г) Корректное множество определений D. Чтобы понять семантику такой системы, нужно знать, как вычислять $f:x$ для любой функции f и любого объекта x, принадлежащих этой системе. Для f имеются четыре возможности:

(1) f является примитивной функцией;
(2) f является функциональной формой;
(3) в D имеется одно определение, **Def** $f \equiv r$;
(4) ни один из предыдущих вариантов не имеет места.

Если f — примитивная функция, то мы располагаем ее определением и знаем, как ее применять. Если f — функциональная форма, то определение формы содержит информацию о том, как вычислять $f:x$ в терминах параметров формы, что может быть сделано при дальнейшем использовании этих правил. Если функция f определена согласно **Def** $f \equiv r$, как в (3), то для отыскания $f:x$ мы вычисляем $r:x$, а это можно сделать дальнейшим применением данных правил. Если ни один из рассмотренных вариантов не имеет места, то $f:x \equiv \bot$. Разумеется, работа по этим правилам может не завершиться при некоторых f и x; в таком случае мы присваиваем значение $f:x \equiv \bot$.

11.3. ПРИМЕРЫ ФУНКЦИОНАЛЬНЫХ ПРОГРАММ

В следующих примерах иллюстрируется функциональный стиль программирования. Поскольку этот стиль незнаком многим читателям, то в их восприятии на первых порах может возникнуть путаница. Важно помнить, что никакая часть определения функции не является результатом как таковым. Напротив, каждая часть представляет собой *функцию*, которую нужно применить к некоему аргументу, чтобы получить результат.

11.3.1. Факториал

Def $!\equiv \mathrm{eq}\,0 \to \bar{1};\ \times \circ [\mathrm{id}, !\circ \mathrm{subl}]$

где

Def $\mathrm{eq}\,0 \equiv \mathrm{eq} \circ [\mathrm{id}, \bar{0}]$

Def $\mathrm{sub}\,1 \equiv - \circ [\mathrm{id}, \bar{1}]$

Здесь приводятся некоторые из промежуточных результатов, которые система ФП получила бы при вычислении $!:2$:

$!:2 \Rightarrow (\mathrm{eq}\,0 \to \bar{1};\ \times \circ [\mathrm{id},\ !\circ \mathrm{subl}]) : 2 \Rightarrow \times \circ (\mathrm{id}, !\circ \mathrm{subl}] : 2$

$\Rightarrow \times:\langle \mathrm{id}:2, !\circ \mathrm{sub}\,1:2\rangle \Rightarrow \times:\langle 2, !:1\rangle$
$\Rightarrow \times:\langle 2, \times:\langle 1, !:0\rangle\rangle$
$\Rightarrow \times:\langle 2, \times:\langle 1, \overline{1}:0\rangle\rangle \Rightarrow \times:\langle 2, \times:\langle 1, 1\rangle\rangle \Rightarrow \times:\langle 2, 1\rangle \Rightarrow 2.$

В разд. 12 мы увидим, как теоремы алгебры программ ФП могут использоваться для доказательства того, что ! является факториальной функцией.

11.3.2. Внутреннее произведение (IP). Мы уже видели ранее, как работает это описание.

Def $\mathrm{IP} \equiv (/+)\circ(\alpha\times)\circ\mathrm{trans}$

11.3.3. Умножение матриц (MM). Эта программа перемножения матриц порождает произведение любой пары (m, n) сопоставимых матриц, причем всякая матрица m представляется как последовательность ее строк:

$m = \langle m_1, \ldots, m_r\rangle,$

где

$m_i = \langle m_{i1}, \ldots, m_{is}\rangle$ при $i = 1, \ldots, r.$

Программа состоит из четырех этапов, читаемых справа налево; каждый из них, начиная с [1, trans∘2], применяется по очереди к результату предыдущего этапа. Если аргумент — это $\langle m, n\rangle$, то первый этап дает

$\langle m, n'\rangle,$

где $n' = \mathrm{trans}:n$. Второй этап дает

$\langle\langle m_1, n'\rangle, \ldots, \langle m_r, n'\rangle\rangle,$

где m_i — это строки из m. Третий этап $\alpha\,\mathrm{distl}$ дает

$\langle \mathrm{distl}:\langle m_1, n'\rangle, \ldots, \mathrm{distl}:\langle m_r, n'\rangle\rangle = \langle p_1, \ldots, p_r\rangle,$

где

$p_i = \mathrm{distl}:\langle m_i, n'\rangle = \langle\langle m_i, n_1'\rangle, \ldots, \langle m_i, n_s'\rangle\rangle$ при $i = 1, \ldots, r$

и n_j' — это j-й столбец из n (j-я строка из n'). Таким образом, p_i, последовательность пар строк и столбцов, соответствует i-й строке произведения. Оператор $\alpha\alpha\mathrm{IP}$, или $\alpha(\alpha\mathrm{IP})$, вызывает применение $\alpha\mathrm{IP}$ к каждой последовательности p_i, что, в свою очередь, влечет за собой применение IP к каждой паре строки и столбца в каждой p_i. Поэтому результатом последнего этапа является последовательность строк, составляющая произведение матриц. Если матрица не прямоугольная, или длина строки матрицы m отличается от длины столбца матрицы n, или если какой-то элемент из m или из n не является числом, то результат равен $\perp$.

Эта программа ММ не именует свои аргументы или какие-либо промежуточные результаты; она не содержит ни переменных, ни циклов, ни операторов управления, ни описаний процедур; в ней нет инструкций инициализации; по своей природе она не является пословной, она иерархически конструируется из более простых компонентов, использует универсальные формы и операторы ведения внутреннего хозяйства (в том числе αf, distl, distr, trans); является совершенно общей; порождает $\perp$ всякий раз, когда ее аргумент оказывается в каком-то смысле неприемлемым; не вносит необязательных ограничений на порядок вычисления (все применения к парам строк и столбцов могут выполняться параллельно или в любом порядке) и с использованием алгебраических законов (см. ниже) может быть преобразована в более «эффективные» или более «наглядные» программы (например, в рекурсивно описанную программу). Ни одним из этих свойств не обладает типичная программа фон Неймана для перемножения матриц.

Несмотря на свою непривычную, а поэтому озадачивающую форму, в отличие от большинства программ программа ММ описывает существенные операции умножения матриц без чрезмерно жесткого определения процесса или затемнения его частей. Поэтому из нее можно получать формальными преобразованиями много непосредственно выполнимых программ. Для компьютеров фон Неймана эта программа неэффективна по самой своей природе (в отношении использования памяти), но из нее можно вывести эффективные модификации, и можно представить себе реализации систем ФП, которые позволили бы выполнять ММ без расточительного использования памяти, предполагаемого приведенной здесь ее формой. Вопросы эффективности выходят за рамки проблематики этой статьи. Я позволю себе только заметить, что поскольку язык прост и не предписывает какой-либо привязки переменных лямбда-типа к данным, то, возможно, лучше было бы, если бы система оказалась способной выполнять некий вид «ленивого» вычисления [9, 10] и контролировать управление данными более эффективно, чем это удается в системах, основанных на лямбда-исчислении.

11.4. ЗАМЕЧАНИЯ О СИСТЕМАХ ФП

11.4.1. Системы ФП как язык программирования. Системы ФП столь скудны, что для некоторых читателей может оказаться затруднительной их интерпретация как языков программирования. При такой интерпретации функция f является программой, объект x представляет собой содержимое памяти, а $f:x$ — это содержимое памяти после активации программы f, когда в памяти находится x. Множество определений представляет со-

бой библиотеку программ. Обеспечиваемые системой примитивные функции и функциональные формы являются базовыми операторами конкретного языка программирования. Итак, в зависимости от выбора примитивных функций и функциональных форм структура ФП обеспечивает большой класс языков с различными стилями и возможностями. Ассоциированная с каждым из этих языков алгебра программ зависит от конкретного набора функциональных форм. Представленные в этой статье примитивные функции, функциональные формы и программы заключают в себе попытку разработать один из таких возможных стилей.

11.4.2. Ограничения систем ФП. Системы ФП характеризуются рядом ограничений. Например, заданная система ФП представляет собой фиксированный язык, она не является исторически чувствительной: ни одна программа не в состоянии изменить библиотеку программ. Она способна интерпретировать входные и выходные данные только в том смысле, что x является входом, а $f:x$ — это выход. Если имеется слабое множество примитивных функций и функциональных форм, то может оказаться, что удается выразить не всякую вычислимую функцию.

Система ФП не может вычислять программу, потому что выражения функций не являются объектами. Нельзя также описывать новые функциональные формы в рамках системы ФП. (Оба этих ограничения устраняются в формальных системах функционального программирования (ФФП), в которых объекты «представляют» функции.) Так, ни одна система ФП не может включать функцию apply, такую, что

$$\text{apply}:\langle x, y\rangle \equiv x:y$$

потому что в левой части x — это объект, а в правой части x — это функция. (Заметим, что мы тщательно следили, чтобы множества символов функций и символов объектов были различными и непересекающимися; так, 1 является символом функции, а *1* — объектом.)

Основное ограничение системы ФП состоит в том, что они исторически нечувствительны. Поэтому их следует как-то расширить, прежде чем они смогут начать приносить практическую пользу. Обсуждение таких расширений приводится в разделах о системах ФФП и АСПС (разд. 13 и 14).

11.4.3. Выразительная сила систем ФП. Предположим, что две ФП-системы FP_1 и FP_2 обладают одинаковыми множествами объектов и одинаковыми множествами примитивных функций, но множество функциональных форм из FP_1 включает соответствующее множество из FP_2, не совпадая с ним. Предпо-

ложим также, что обе системы могут выражать все вычислимые функции на объектах. Тем не менее мы можем сказать, что система FP_1 более выразительна, чем система FP_2, поскольку всякое выражение функции в FP_2 может быть продублировано в FP_1, но за счет использования функциональной формы, не принадлежащей к FP_2, система FP_1 способна выражать некоторые функции более прямо и легко, чем система FP_2.

Я полагаю, что приведенные выше соображения могут быть развиты в теорию выразительной силы языков, согласно которой язык A оказался бы *более выразительным*, чем язык B, при следующих грубо сформулированных условиях. Во-первых, формируем все возможные функции любых типов на A, применяя все существующие функции к объектам и друг к другу всеми возможными способами до тех пор, пока нельзя будет сформировать никакую новую функцию какого-либо типа. (Множество объектов является типом. Множество непрерывных функций $[T \to U]$ из типа T в тип U является типом. Если $f \in [T \to U]$ и $t \in T$, то ft из U может быть сформировано применением f к t.) Делаем то же самое для языка B. Затем сравниваем любой тип из A с соответствующим типом из B. Если для любого типа верно, что тип из A включает соответствующий тип из B, то язык A более выразителен, чем язык B (или столь же выразителен). Если некоторый тип функции из A не имеет эквивалента в языке B[1], то языки A и B несравнимы по выразительной мощности.

11.4.4. Преимущества системы ФП. Системы ФП значительно проще, чем традиционные языки и языки, основанные на лямбда-исчислении, в основном по той причине, что в них употребляется только самая простая фиксированная система именования (именование функции в ее определении) с простым фиксированным правилом подстановки функции вместо ее имени. Поэтому в них отсутствует сложность как систем именования в традиционных языках, так и правил подстановок из лямбда-исчисления. Системы ФП допускают определение различных систем именования (см. разд. 13.3.4 и 14.7) для разных целей. От них не требуется сложность, поскольку многие программы могут работать совсем без них. Более существенно, что эти системы интерпретируют имена как функции, которые могут комбинироваться с другими функциями и не требуют специального подхода.

Системы ФП позволяют освободиться от традиционного по-

[1] И наоборот, некий тип из B не имеет эквивалента в A. — *Прим. перев.*

словного программирования еще в большей степени, чем язык APL [12] (наиболее успешный на сегодняшний день подход к данной проблеме в рамках структуры фон Неймана), потому что они обеспечивают более мощный набор функциональных форм в едином мире выражений. Они позволяют развивать методы более высокого уровня, пригодные для размышлений над программами, манипулирования ими и их написания.

12. АЛГЕБРА ПРОГРАММ ДЛЯ СИСТЕМ ФП

12.1. ВВЕДЕНИЕ

Описываемая ниже алгебра программ является любительской работой в области алгебры, и я хочу показать, что любители могут успешно играть в эту игру и получать от нее удовлетворение, а также что эта игра не требует глубокого понимания логики и математики. Несмотря на свою простоту, она может помогать понимать и доказывать факты о программах систематическим, в каком-то смысле математическим способом.

Пока что доказательство корректности программ требует знания довольно сложных разделов математики и логики, свойств полных частично упорядоченных множеств, непрерывных функций, наименьших фиксированных точек функционалов, исчисления высказываний первого порядка, преобразователей предикатов, слабейших предусловий (здесь упомянуты лишь некоторые темы, необходимые для нескольких подходов к доказательству корректности программ). Эти темы оказывались очень полезными для профессионалов, которые сделали своим основным занятием изобретение методов доказательства; они опубликовали уйму блестящих работ по этой теме, начиная с исследований Маккарти и Флойда до более недавних работ Барстелла, Дейкстры, Манны и его сотрудников, Милнера, Морриса, Рейнольдса и многих других. Большая часть этих исследований основывается на фундаменте, заложенном Д. Скоттом (денотационные семантики) и Ч. Хоаром (аксиоматические семантики). Но по своему теоретическому уровню они недоступны большинству любителей, работающих вне этой специализированной области.

Если программисту средней квалификации требуется доказать корректность своей программы, ему понадобятся гораздо более простые методы, чем те, которые до сих пор развивали теоретики. Излагаемая ниже алгебра программ может послужить одной из отправных точек для такой дисциплины доказательств, и в сочетании с текущей работой по алгебраическим манипуляциям она может также способствовать построению основы для частичной автоматизации этой дисциплины.

Одно из преимуществ этой алгебры по сравнению с другими методами доказательств состоит в том, что программист может использовать свой язык программирования в качестве языка вывода доказательств, а не вынужден формулировать доказательства в отдельной логической системе, в которой можно только *рассуждать* о его программе.

Сердцевину алгебры программ составляют законы и теоремы, утверждающие, что одно выражение функции эквивалентно другому. Например, закон $[f, g]\circ h \equiv [f\circ h, g\circ h]$ гласит, что конструкция из f и g (в композиции с h) представляет собой такую же функцию, как конструкция из (f в композиции с h) и (g в композиции с h) вне зависимости от того, каковы функции f, g и h. Такие законы легко понимаются, легко обосновываются, они мощны и удобны для практического применения. Однако мы хотим также применять такие законы для решений уравнений, в которых «неизвестная» функция появляется в обеих частях уравнения. Проблема состоит в том, что, если f удовлетворяет некоторому такому уравнению, часто случается, что и некое обобщение f' для f будет также удовлетворять тому же уравнению. Таким образом, для того чтобы обеспечить однозначность решений таких уравнений, мы должны потребовать, чтобы основания алгебры программ (с использованием введенного Скоттом понятия наименьших фиксированных точек непрерывных функционалов) гарантировали нам, что решения, получаемые алгебраическими манипуляциями, действительно являются наименьшими в этом смысле, а следовательно, единственными решениями.

Наша цель состоит в том, чтобы разработать основания алгебры программ, свободные от теоретических премудростей, и тем самым позволить программисту использовать простые алгебраические законы и одну или две теоремы из этих оснований для решения задач и построения доказательств точно таким же механическим образом, как мы решаем алгебраические задачи в высшей школе, при этом ничего не зная о наименьших фиксированных точках или преобразователях предикатов.

Возникает одна конкретная проблема, относящаяся к этим основаниям: если заданы уравнения вида

$$f \equiv p_0 \rightarrow q_0; \ldots; p_i \rightarrow q_i;\ E_i(f) \qquad (1)$$

где p_i и q_i — это функции, не включающие f, а $E_i(f)$ — функциональное выражение, включающее f, то законы алгебры часто позволяют формально «расширить» это уравнение еще на одну «фразу» за счет вывода

$$E_i(f) \equiv p_{i+1} \rightarrow q_{i+1};\ E_{i+1}(f) \qquad (2)$$

что при замене $E_i(f)$ в (1) на правую часть из (2) порождает

$$f \equiv p_0 \to q_0; \ldots; p_{i+1} \to q_{i+1}; \; E_{i+1}(f) \qquad (3)$$

Такое формальное расширение может продолжаться без ограничений. В таком случае из оснований должен следовать ответ на вопрос: когда наименьшая f, удовлетворяющая (1), может быть представлена бесконечным разложением

$$f \equiv p_0 \to q_0; \ldots; p_n \to q_n; \ldots \qquad (4)$$

в котором последняя фраза, включающая f, была опущена, так что теперь мы имеем решение с правой частью, свободной от f? Такие решения полезны в двух отношениях: во-первых, они дают доказательства «завершимости» в том смысле, что вследствие (4) $f:x$ определено тогда и только тогда, когда существует значение n, при котором для всякого $i<n$ имеем $p_i:x=F$ и $p_n:x=T$, и определено $q_n:x$. Во-вторых, формула (4) дает последовательное описание функции f, которое часто может прояснить поведение функции.

Приводимые в последующем разделе основания алгебры — это скромный шаг в сторону сформулированной выше цели. Для ограниченного класса уравнений соответствующая «теорема линейного расширения» дает полезный ответ относительно того, когда можно перейти от неопределенно продолжаемых уравнений типа (1) к бесконечным разложениям типа (4). Для более обширного класса уравнений более общая «теорема разложения» дает менее полезный ответ на сходные вопросы. К счастью, можно найти более общие теоремы, покрывающие дополнительные классы уравнений. Но в настоящее время достаточно знать только следствия этих двух простых фундаментальных теорем, чтобы понимать теоремы и примеры, представленные в этом разделе.

Результаты подраздела об основаниях обобщаются в отдельном, более раннем подразделе, озаглавленном «теоремы разложения», без ссылок на концепции фиксированных точек. Сам подраздел об основаниях помещен позже и может быть опущен читателями, не желающими углубляться в этот предмет.

12.2. ЗАКОНЫ АЛГЕБРЫ ПРОГРАММ

В алгебре программ для системы ФП переменные варьируются в пределах множества функций системы. «Операциями» алгебры являются функциональные формы системы. Так, например, $[f, g]\circ h$ — это выражение такой алгебры для описанной выше системы ФП, в которой переменные f, g и h обозначают произвольные функции этой системы. И

$[f, g]\circ h \equiv [f\circ h, g\circ h]$

представляет собой закон алгебры, который гласит, что, какие бы функции не были выбраны для f, g и h, функция в левой части не отличается от функции в правой части. Таким образом, этот алгебраический закон только формулирует по-иному следующее утверждение относительно любой системы ФП, которая включает функциональные формы $[f, g]$ и $f\circ g$.

Утверждение. *Для всяких функций f и g и для любых объектов x верно* $([f, g]\circ h) : x \equiv [f\circ h, g\circ h] : x$.

Доказательство.

$$\begin{aligned}([f, g]\circ h) : x &= [f, g] : (h : x) && \text{по определению композиции}\\ &= \langle f : (h : x), g : (h : x)\rangle && \text{по определению конструкции}\\ &= \langle (f\circ h) : x, (g\circ h) : x\rangle && \text{по определению композиции}\\ &= [f\circ h, g\circ h] : x && \text{по определению конструкции}\ \square\end{aligned}$$

Области действия некоторых законов меньше, чем область всех объектов. В частности, $1\circ[f, g] \equiv f$ не верно для объектов x, таких, что $g : x = \bot$. Мы записываем

$$\text{defined}\circ g \rightarrow\rightarrow 1\circ[f, g] \equiv f$$

чтобы указать, что закон (или теорема) справа справедлив в области объектов x, для которых $\text{defined}\circ g : x = T$, где

Def $\text{defined} \equiv \overline{T}$

т. е. $\text{defined} : x \equiv x = \bot \rightarrow \bot;\ T$. В общем случае мы будем записывать *ограниченное функциональное уравнение*

$$p \rightarrow\rightarrow f \equiv g$$

которое означает, что для любого объекта x всякий раз, когда $p : x = T$, верно $f : x = g : x$.

Обычная алгебра имеет дело с двумя операциями, сложением и вычитанием, и нуждается в немногих законах. Алгебра программ имеет дело с большим числом операций (функциональных форм) и поэтому нуждается в большем количестве законов.

Каждый из следующих законов требует, чтобы выполнялось соответствующее утверждение. Интересующийся читатель легко найдет большинство доказательств таких утверждений (два из них приведены ниже). Определим сначала обычное упорядочение функций и эквивалентность в терминах этого упорядочения:

Определение. $f \leqslant g$ тогда и только тогда, когда для всех объектов x либо $f : x = \bot$, либо $f : x = g : x$

Определение. $f \equiv g$ тогда и только тогда, когда $f \leqslant g$ и $g \leqslant f$.

Легко убедиться в том, что $\leqslant$ является частичным упорядочением; $f \leqslant g$ означает, что g является обобщением для f, а $f \equiv g$ тогда и только тогда, когда $f:x = g:x$ для всяких объектов x. Приведем теперь список алгебраических законов, перечисляемых по парам основных функциональных форм.

I. Композиция и конструкция

I.1 $[f_1, \ldots, f_n] \circ g \equiv [f_1 \circ g, \ldots, f_n \circ g]$

I.2 $\alpha f \circ [g_1, \ldots, g_n] \equiv [f \circ g_1, \ldots, f \circ g_n]$

I.3 $/f_0[g_1, \ldots, g_n] \equiv f \circ [g_1, /f_0[g_2, \ldots, g_n]]$ при $n \geqslant 2$.
$\equiv f_0[g_1, f \circ [g_2, \ldots, f \circ [g_{n-1}, g_n] \ldots]]$
$/f \circ [g] \equiv g$

I.4 $f \circ [\bar{x}, g] \equiv (\mathrm{bu}\ f\ x) \circ g$

I.5 $1 \circ [f_1, \ldots, f_n] \leqslant f_1$
$s \circ [f_1, \ldots, f_s, \ldots, f_n] \leqslant f_s$ для любого селектора s, $s \leqslant n$
$\mathrm{defined} \circ f_i$ (при всех $i \neq s$, $1 \leqslant i \leqslant n$) $\rightarrow\rightarrow s \circ [f_1, \ldots, f_n] \equiv f_s$

I.5.1 $[f_1 \circ 1, \ldots, f_n \circ n] \circ [g_1, \ldots, g_n] \equiv [f_1 \circ g_1, \ldots, f_n \circ g_n]$

I.6 $\mathrm{tl} \circ [f_1] \leqslant \varnothing$ и $\mathrm{tl} \circ [f_1, \ldots, f_n] \leqslant [f_2, \ldots, f_n]$ при $n \geqslant 2$
$\mathrm{defined} \circ f_1 \rightarrow\rightarrow \mathrm{tl} \circ [f_1] \equiv \varnothing$
и $\mathrm{tl} \circ [f_1, \ldots, f_n] \equiv [f_2, \ldots, f_n]$ при $n \geqslant 2$

I.7 $\mathrm{distl} \circ [f, [g_1, \ldots, g_n]] \equiv [[f, g_1], \ldots, [f, g_n]]$.
$\mathrm{defined} \circ f \rightarrow\rightarrow \mathrm{distl} \circ [f, \overline{\varnothing}] \equiv \overline{\varnothing}$

Аналогичный закон справедлив для дистрибутивности справа (distr).

I.8 $\mathrm{apndl} \circ [f, [g_1, \ldots, g_n]] \equiv [f, g_1, \ldots, g_n]$
$\mathrm{null} \circ g \rightarrow\rightarrow \mathrm{apndl} \circ [f, g] \equiv [f]$

и так далее, для apndr, обращения (reverse), поворота (rotl) и др.

I.9 $[\ldots, \overline{\perp}, \ldots] \equiv \overline{\perp}$

I.10 $\mathrm{apndl} \circ [f \circ g, \alpha f \circ h] \equiv \alpha f \circ \mathrm{apndl} \circ [g, h]$

I.11 $\mathrm{pair} \,\&\, \mathrm{not} \circ \mathrm{null} \circ 1 \rightarrow\rightarrow \mathrm{apndl} \circ [[1 \circ 1, 2], \mathrm{distr} \circ [\mathrm{tl} \circ 1, 2]] \equiv \mathrm{distr}$
где $f \,\&\, g \equiv \mathrm{and} \circ [f, g]$; $\mathrm{pair} \equiv \mathrm{atom} \rightarrow \bar{F}$; $\mathrm{eq} \circ [\mathrm{length}, \overline{2}]$

II. Композиция и условие (ассоциирующиеся справа скобки опущены) (Закон II.2 указан в [16, p. 493].)

II.1 $(p \rightarrow f; g) \circ h \equiv p \circ h \rightarrow f \circ h; g \circ h$

II.2 $h \circ (p \rightarrow f; g) \equiv p \rightarrow h \circ f; h \circ g$

II.3 $\mathrm{or} \circ [q, \mathrm{not} \circ q] \rightarrow\rightarrow \mathrm{and} \circ [p, q] \rightarrow f; \mathrm{and} \circ [p, \mathrm{not} \circ q] \rightarrow g; h$
$= p \rightarrow (q \rightarrow f; g); h$

II.3.1 $p \rightarrow (p \rightarrow f; g); h \equiv p \rightarrow f; h$

III. Композиция и смешение

III.1 $\bar{x}\circ f \leqslant \bar{x}$
$\text{defined}\circ f \to\to \bar{x}\circ f = \bar{x}$

III.1.1 $\overline{\perp}\circ f = f\circ\overline{\perp} = \overline{\perp}$

III.2 $f\circ\text{id} = \text{id}\circ f = f$

III.3 $\text{pair}\to\to 1\circ\text{distr} = [1\circ 1, 2]$ также: $\text{pair}\to\to 1\circ t = 2$ и т. д.

III.4 $\alpha(f\circ g) = \alpha f\circ\alpha g$

III.5 $\text{null}\circ g \to\to \alpha f\circ g = \overline{\varnothing}$

IV. Условие и конструкция

IV.1 $[f_1, \ldots, (p\to g; h), \ldots, f_n]$
$= p\to [f_1, \ldots, g, \ldots, f_n]; [f_1, \ldots, h, \ldots, f_n]$

IV.1.1 $[f_1, \ldots, (p_1\to g_1, \ldots, p_n\to g_n; h), \ldots, f_m]$
$= p_1\to[f_1, \ldots, g_1, \ldots, f_m);$
$\ldots; p_n\to [f_1, \ldots, g_n, \ldots, f_m]; [f_1, \ldots, h, \ldots, f_m]$

Этим завершается данный список алгебраических законов; он никоим образом не является исчерпывающим; существуют многие другие законы.

Доказательства двух законов

Мы приводим доказательства справедливости утверждения для законов I.10 и I.11, которые немного сложнее, чем большинство остальных.

Утверждение 1

$\text{apndl}\circ[f\circ g, \alpha f\circ h] = \alpha f\circ\text{apndl}\circ[g, h]$

Доказательство. Мы покажем, что для любого объекта x обе указанные выше функции порождают одинаковые результаты.

Случай 1. $h:x$ — это не последовательность и не $\varnothing$. Тогда обе части при применении к x порождают $\perp$.

Случай 2. $h:x = \varnothing$. Тогда

$\text{apndl}\circ[f\circ g, \alpha f\circ h]:x = \text{apndl}:\langle f\circ g:x, \varnothing\rangle = \langle f:(g:x)\rangle$
$\alpha f\circ\text{apndl}\circ[g, h]:x = \alpha f\circ\text{apndl}:\langle g:x, \varnothing\rangle = \alpha f:\langle g:x\rangle$
$= \langle f:(g:x)\rangle$

Случай 3. $h:x = \langle y_1, \ldots, y_n\rangle$. Тогда

$\text{apndl}\circ[f\circ g, \alpha f\circ h]:x = \text{apndl}:\langle f\circ g:x, \alpha f:\langle y_1, \ldots, y_n\rangle\rangle$
$= \langle f:(g:x), f:y_1, \ldots, f:y_n\rangle$
$\alpha f\circ\text{apndl}\circ[g, h]:x = \alpha f\circ\text{apndl}:\langle g:x, \langle y_1, \ldots, y_n\rangle\rangle$
$= \alpha f:\langle g:x, y_1, \ldots, y_n\rangle$
$= \langle f:(g:x), f:y_1, \ldots, f:y_n\rangle$

Утверждение 2

$\text{Pair}\,\&\circ\text{not}\circ\text{null}\circ 1 \to\to \text{apndl}\circ[[1^2, 2], \text{distr}\circ[\text{tl}\circ 1, 2]]] = \text{distr},$

где $f \& g$ является функцией: $\text{and} \circ [f, g]$, $\text{and} f^2 \equiv f \circ f$.

Доказательство. Мы покажем, что обе части порождают одинаковый результат при их применении к любой паре $\langle x, y \rangle$, где $x \neq \varnothing$ согласно сформулированному ограничению.

Случай 1. x является атомом или $\perp$. Тогда $\text{distr} : \langle x, y \rangle = \perp$, потому что $x \neq \varnothing$. Левая часть тоже порождает $\perp$ при применении к $\langle x, y \rangle$, поскольку $\text{tl} \circ 1 : \langle x, y \rangle = \perp$ и все функции сохраняют $\perp$.

Случай 2. $x = \langle x_1, \ldots, x_n \rangle$. Тогда

$$\begin{aligned}
&\text{apndl} \circ [[1^2, 2], \text{distr} \circ [\text{tl} \circ 1, 2]] : \langle x, y \rangle \\
&= \text{apndl} : \langle\langle 1 : x, y \rangle, \text{distr} : \langle \text{tl} : x, y \rangle\rangle \\
&= \text{apndl} : \langle\langle x_1, y \rangle, \varnothing \rangle = \langle\langle x_1, y \rangle\rangle \text{ если } \text{tl} : x = \varnothing \\
&= \text{apndl} : \langle\langle x_1, y \rangle, \langle\langle x_2, y \rangle, \ldots, \langle\langle x_n, y \rangle\rangle\rangle\rangle \text{ если } \text{tl} : x \neq \varnothing \\
&= \langle\langle x_1, y \rangle, \ldots, \langle x_n, y \rangle\rangle \\
&= \text{distr} : \langle x, y \rangle.
\end{aligned}$$

□

12.3. ПРИМЕР: ЭКВИВАЛЕНТНОСТЬ ДВУХ ПРОГРАММ УМНОЖЕНИЯ МАТРИЦ

Мы рассматривали ранее программу умножения матриц:

Def $MM = \alpha\alpha\, \text{IP} \circ \alpha\, \text{distr} \circ [1, \text{trans} \circ 2]$.

Теперь мы покажем, что ее начальный сегмент MM', где

Def $MM' \equiv \alpha\alpha\, \text{IP} \circ \text{distl} \circ \text{distr}$

может быть описан рекурсивно. (MM' «перемножает» пару матриц после того, как вторая матрица транспонирована. Заметим, что MM' в отличие от MM дает $\perp$ для любых аргументов, которые не образуют пар.) Иначе говоря, мы покажем, что MM' удовлетворяет следующему уравнению, рекурсивно описывающему ту же функцию (на парах):

$f \equiv \text{null} \circ 1 \rightarrow \overline{\varnothing};\ \text{apndl} \circ [\alpha\, \text{IP} \circ \text{distl} \circ [1 \circ 1, 2], f \circ [\text{tl} \circ 1, 2]]$.

Наше доказательство примет форму демонстрации того, что следующая функция R, где

Def $R \equiv \text{null} \circ 1 \rightarrow \overline{\varnothing};\ \text{apndl} \circ [\alpha\, \text{IP} \circ \text{distl} \circ [1 \circ 1, 2], MM' \circ [\text{tl} \circ 1, 2]]$

при любых парах $\langle x, y \rangle$ эквивалентна функции MM'. Функция R «умножает» две матрицы, когда первая матрица содержит более чем 0 строк; вычисляется первая строка «произведения» (при помощи $\alpha\, \text{IP} \circ \text{distl} \circ [1 \circ 1, 2]$) и присоединяется к «произведению» остатка первой матрицы и второй матрицы. Таким образом, нам нужна теорема

$\text{pair} \rightarrow\rightarrow MM \equiv R$,

а из этого сразу следует

$MM \equiv MM' \circ [1, \text{trans} \circ 2] \equiv R \circ [1, \text{trans} \circ 2]$, где

Def $\text{pair} \equiv \text{atom} \to \overline{F}; \text{eq} \circ [\text{length}, \overline{2}]$.

Теорема: $\text{pair} \to\to MM' = R$,

где

Def $MM' \equiv \alpha\alpha\, \text{IP} \circ \alpha\, \text{distl} \circ \text{distr}$

Def $R \equiv \text{null} \circ 1 \to \overline{\varnothing}; \text{apndl} \circ [\alpha\, \text{IP} \circ \text{distl} \circ [1^2, 2], MM' \circ [\text{tl} \circ 1, 2]]$

Доказательство.

Случай 1. $\text{pair} \,\&\, \text{null} \circ 1 \to\to MM' \equiv R$

$\text{pair} \,\&\, \text{null} \circ 1 \to\to R \equiv \varnothing$ по определению R

$\text{pair} \,\&\, \text{null} \circ 1 \to\to MM' \equiv \varnothing$

так как $\text{distr} : \langle \varnothing, x \rangle = \varnothing$ по определению distr

и $\alpha f : \varnothing = \varnothing$ по определению «Применить ко всем»

Таким образом, $\alpha\alpha\, \text{IP} \circ \alpha\, \text{distl} \circ \text{distr} : \langle \varnothing, x \rangle = \varnothing$.

Поэтому $\text{pair} \,\&\, \text{null} \circ 1 \to\to MM' \equiv R$.

Случай 2. $\text{pair} \,\&\, \text{not} \circ \text{null} \circ 1 \to\to MM' \equiv \text{R}$.

$\text{pair} \,\&\, \text{not} \circ \text{null} \circ 1 \to\to \text{R} \equiv \text{R}'$ по определению R и R′, где (1)

Def $\text{R}' \equiv \text{apndl} \circ [\alpha\, \text{IP} \circ \text{distl} \circ [1^2, 2], MM \circ [\text{tl} \circ 1, 2]]$.

Заметим, что

$\text{R}' \equiv \text{apndl} \circ [f \circ g, \alpha f \circ h]$,

где

$f \equiv \alpha\, \text{IP} \circ \text{distl}$

$g \equiv [1^2, 2]$

$h \equiv \text{distr} \circ [\text{tl} \circ 1, 2]$

$\alpha f \equiv \alpha(\alpha\, \text{IP} \circ \text{distl}) \equiv \alpha\alpha\, \text{IP} \circ \alpha\, \text{distl}$ (согласно III.4). (2)

Таким образом, согласно 1.10,

$R' \equiv \alpha f \circ \text{apndl} \circ [g, h]$. (3)

Теперь $\text{apndl} \circ [g, h] \equiv \text{apndl} \circ [[1^2, 2], \text{distr} \circ [\text{tl} \circ 1, 2]]$, поэтому в силу I.11

$\text{pair} \,\&\, \text{not} \circ \text{null} \circ 1 \to\to \text{apndl} \circ [g, h] \equiv \text{distr}$. (4)

Итак, мы имеем, согласно (1), (2), (3) и (4),

$\text{pair} \,\&\, \text{not} \circ \text{null} \circ 1 \to\to \text{R} \equiv \text{R}'$

$\equiv \alpha f \circ \text{distr} \equiv \alpha\alpha\, \text{IP} \circ \alpha\, \text{distl} \circ \text{distr} \equiv MM'$.

Случаи 1 и 2, взятые вместе, доказывают теорему. □

12.4. ТЕОРЕМЫ РАЗЛОЖЕНИЙ

В следующих подразделах мы займемся «решением» некоторых простых уравнений (здесь под «решением» мы понимаем отыскание «наименьшей» функции, которая удовлетворяет уравнению). Для этого нам понадобятся следующие понятия и результаты, выводимые из дальнейшего подраздела об основаниях алгебры, где появляются их доказательства.

12.4.1. Разложение. Предположим, что у нас имеется уравнение вида

$$f \equiv E(f) \qquad \text{(E1)}$$

где $E(f)$ — выражение, включающее f. Предположим далее, что имеется бесконечная последовательность функций f_i для $i=0, 1, 2, \ldots$, каждая из которых имеет следующий вид:

$$f_0 \equiv \overline{\perp}$$
$$f_{i+1} \equiv p_0 \to q_0; \ldots; p_i \to q_i; \overline{\perp} \qquad \text{(E2)}$$

где p_i' и q_i' — это конкретные функции, так что E обладает свойством

$$E(f_i) \equiv f_{i+1} \text{ при } i=0, 1, 2, \ldots . \qquad \text{(E3)}$$

Тогда мы говорим, что выражение E разложимо и имеет f_i в качестве *аппроксимирующих функций.*

Если E разложимо и обладает аппроксимирующими функциями, как в (E2), и если f является решением для (E1), то f можно записать как бесконечное разложение

$$f \equiv p_0 \to q_0; \ldots; p_n \to q_n; \ldots \qquad \text{(E4)}$$

означающее, что для любого x $f:x \neq \perp$ тогда и только тогда, когда имеется значение $n \geqslant 0$, такое, что (а) $p_i : x = F$ для всех $i<n$, (б) $p_n : x = T$, (в) $q_n : x \neq \perp$. Если $f : x \neq \perp$, то $f : x = q_n : x$ для этого значения n. (Сказанное выше является следствием «теоремы о разложении».)

12.4.2. Линейное разложение. Существует более удобное средство решения некоторых уравнений, оно применимо, когда для любой функции h

$$E(h) \equiv p_0 \to q_0; E_1(h) \qquad \text{(LE1)}$$

и существуют p_i и q_i, такие, что

$$E_1(p_i \to q_i; h) \equiv p_{i+1} \to q_{i+1}; E_1(h) \text{ при } i=0, 1, 2, \ldots \qquad \text{(LE2)}$$

и $E_1(\overline{\perp}) \equiv \overline{\perp}$. (LE3)

При этих условиях выражение E называется *линейно разложимым.*

Если оно таково и f является решением для

$$f \equiv E(f) \qquad \text{(LE4)}$$

то E и f можно переписать как бесконечное разложение

$$f \equiv p_0 \to q_0 ; \ldots ; p_n \to q_n ; \ldots \qquad \text{(LE5)}$$

с применением p_i и q_i, генерируемых по (LE1) и (LE2).

Хотя p_i и q_i из (E4) и (LE5) не являются единственными для данной функции, может оказаться возможным найти дополнительные ограничения, которые придают им однозначность, и в таком случае разложение (LE5) представляло бы собой каноническую форму функции f. Даже без однозначности такие разложения часто позволяют доказывать эквивалентность двух различных функциональных выражений и часто проясняют поведение функции.

12.5. ТЕОРЕМА О РЕКУРСИИ

Используя три из сформулированных выше законов и линейное разложение, можно доказать следующую довольно общую теорему, которая дает разложение, проясняющее многие рекурсивно определенные функции.

Теорема рекурсии. *Пусть* f *является решением для*

$$f \equiv p \to g;\ Q(f) \qquad (1)$$

где

$$Q(k) \equiv h \circ [i,\ k \circ j] \text{ для любой функции } k \qquad (2)$$

и p, g, h, i, j *— это любые заданные функции; тогда*

$$f \equiv p \to g,\ p \circ j \to Q(g); \ldots ; p \circ j^n \to Q^n(g); \ldots \qquad (3)$$

(*где* $Q^n(g)$ *есть* $h \circ [i,\ Q^{n-1}(g) \circ j]$ *и* j^n *есть* $j \circ j^{n-1}$ *для* $n \geqslant 2$) *и*

$$Q^n(g) \equiv /h \circ [i,\ i \circ j,\ \ldots,\ i \circ j^{n-1},\ g \circ j^n]. \qquad (4)$$

Доказательство. Мы доказываем, что $p \to g;\ Q(f)$ линейно разложимо. Пусть p_n, q_n и k суть произвольные функции. Тогда

$$\begin{aligned}
Q(p_n \to q_n;\ k) &\equiv h \circ [i,\ (p_n \to q_n;\ k) \circ j] && \text{по (2)} \\
&\equiv h \circ [i,\ (p_n \circ j \to q_n \circ j;\ k \circ j)\,] && \text{по II.1} \\
&\equiv h \circ (p_n \circ j \to [i,\ q_n \circ j];\ [i,\ k \circ j]) && \text{по IV.1 (5)} \\
&\equiv p_n \circ j \to h \circ [i,\ q_n \circ j];\ h \circ [i,\ k \circ j] && \text{по II.2} \\
&\equiv p_n \circ j \to Q(q_n);\ Q(k) && \text{по (2)}
\end{aligned}$$

Итак, если $p_0 \equiv p$ и $q_0 \equiv q$, то (5) дает $p_1 \equiv p \circ j$ и $q_1 \equiv Q(g)$, а в общем случае дает следующие функции, удовлетворяющие

(LE2):

$p_n \equiv p \circ j^n$ и $q_n \equiv Q^n(g)$. (6)

Наконец,

$$\begin{aligned} Q(\overline{\perp}) &\equiv h \circ [i, \overline{\perp} \circ j] \\ &\equiv h \circ [i, \overline{\perp}] && \text{по III.1.1} \\ &\equiv h \circ \overline{\perp} && \text{по I.9} \\ &\equiv \overline{\perp} && \text{по III.1.1} \end{aligned} \qquad (7)$$

Итак, (5) и (6) обосновывают (LE2), и (7) обосновывает (LE3) при $E_1 \equiv Q$. Если принять $E(f) \equiv f \to g;\ Q(f)$, то мы получаем (LE1); таким образом, выражение E линейно разложимо. Поскольку f является решением для $f \equiv E(f)$, вывод (3) следует из (6) и (LE5). Теперь

$$\begin{aligned} Q^n(g) &\equiv h \circ [i, Q^{n-1}(g) \circ j] \\ &\equiv h \circ [i, h \circ [i \circ j, \ldots, h \circ [i \circ j^{n-1}, g \circ j^n] \ldots]] && \text{по I.1, примененному многократно} \\ &\equiv /h \circ [i, i \circ j, \ldots, i \circ j^{n-1}, g \circ j^n] && \text{по I.3.} \end{aligned} \qquad (8)$$

Результат (8) является вторым выводом (4).

12.5.1. Пример: доказательство корректности для рекурсивной факториальной функции. Пусть f — это решение для

$f \equiv \mathrm{eq}\,0 \to \overline{1};\ \times \circ [\mathrm{id}, f \circ s]$

где

Def $s \equiv -\!\!\circ [id, \overline{1}]$ (вычитание 1).

Тогда f удовлетворяет предположению теоремы рекурсии для $p \equiv \mathrm{eq}\,0$, $g \equiv \overline{1}$, $h \equiv \times$, $i \equiv \mathrm{id}$ и $j \equiv s$.

Поэтому

$f \equiv \mathrm{eq}\,0 \to \overline{1}; \ldots; \mathrm{eq}\,0 \circ s^n \to Q^n(\overline{1}); \ldots$

и

$Q^n(\overline{1}) \equiv /\times \circ [\mathrm{id}, \mathrm{id} \circ s, \ldots, \mathrm{id} \circ s^{n-1}, \overline{1} \circ s^n]$.

Теперь $\mathrm{id} \circ s^k \equiv s^k$ согласно III.2 и $\mathrm{eq}\,0 \circ s^n \to \to \overline{1} \circ s^n \equiv \overline{1}$ согласно III.1, поскольку $\mathrm{eq}\,0 \circ s^n : x$ следует $\mathrm{defined} \circ s^n : x$, а также

$\mathrm{eq}\,0 \circ s^n : x \equiv \mathrm{eq}\,0 : (x - \mathrm{n}) \equiv x = \mathrm{n}$.

Итак, если $\mathrm{eq}\,0 \circ s^n : x = T$, то $x = \mathrm{n}$ и

$Q^n(\overline{1})\ \mathrm{n} = \mathrm{n} \times (\mathrm{n}-1) \times \ldots \times (\mathrm{n} - (\mathrm{n}-1)) \times (\overline{1} : (\mathrm{n} - \mathrm{n})) = \mathrm{n}!$.

Используя эти результаты для $\overline{1}\circ s^n$, $\mathrm{eq}\,0\circ s^n$ и $Q^n(1)$ в предыдущем разложении для f, получаем

$$f:x\equiv x=0\to\overline{1};\ldots;x=n\to n\times(n-1)\times\ldots\times1\times1;\ldots.$$

Таким образом, мы доказали, что f завершается в точности на множестве неотрицательных целых, и поэтому она является факториальной функцией.

12.6. ТЕОРЕМА ОБ ИТЕРАЦИИ

В сущности, это дополнение к теореме о рекурсии. Она дает простое разложение для многих итеративных программ.

Теорема об итерации. *Пусть f — это решение (т. е. наименьшее решение) для $f\equiv p\to g;\ h\circ f\circ k$; тогда*

$$f\equiv p\to g;\ p\circ k\to h\circ g\circ k;\ldots;p\circ k^n\to h^n\circ g\circ k^n;\ldots.$$

Доказательство. Пусть $h'\equiv h\circ 2$, $i'\equiv\mathrm{id}$, $j'\equiv k$, тогда

$$f\equiv p\to g;\ h'\circ[i',f\circ j']$$

поскольку $h\circ2\circ[\mathrm{id},f\circ k]\equiv h\circ f\circ k$ согласно I.5 (id определен всюду, кроме $\perp$, а уравнение справедливо для $\perp$). Итак, теорема о рекурсии дает

$$f\equiv p\to g;\ldots;p\circ k^n\to Q^n(g);\ldots$$

где

$$Q^n(g)\equiv h\circ2\circ[id,\ Q^{n-1}(g)\circ k]$$
$$\equiv h\circ Q^{n-1}(g)\circ k\equiv h^n\circ g\circ k^n \quad \text{по I.5} \qquad \square$$

12.6.1. Пример: доказательство корректности для итеративной факториальной функции. Пусть f — решение для

$$f\equiv\mathrm{eq}\,0\circ1\to2;\ f\circ[s\circ1,\times]$$

где **Def** $s\equiv-\circ[\mathrm{id},\overline{1}]$ (вычитание 1). Мы хотим доказать, что $f:\langle x,1\rangle=x!$ тогда и только тогда, когда x — это неотрицательное целое. Пусть $p\equiv\mathrm{eq}\circ1$, $g\equiv2$, $h\equiv\mathrm{id}$, $k\equiv[s\circ1,\times]$. Тогда $f\equiv p\to g$, $h\circ f\circ k$ и поэтому

$$f\equiv p\to g;\ldots;p\circ k\to g\circ k^n,\ \ldots \qquad (1)$$

в силу теоремы об итерации, так как $h^n\equiv\mathrm{id}$.
Мы хотим показать, что

$$\mathrm{pair}\to\to k^n\equiv[a_n,\ b_n] \qquad (2)$$

верно для любого значения $n\geqslant1$, где

$$a_n\equiv s^n\circ1 \qquad (3)$$

$$b_n \equiv /\times \circ [s^{n-1} \circ 1, \ldots, s \circ 1, 1, 2]. \quad (4)$$

Теперь (2) имеет место для $n \equiv 1$ по определению k. Предположим, что (2) верно для некоторого значения $n \geqslant 1$, и докажем, что это утверждение верно для $n+1$. Имеем

$$\text{pair} \rightarrow\rightarrow k^{n+1} \equiv k \circ k^n \equiv [s \circ 1, \times] \circ [a_n, b_n] \quad (5)$$

поскольку (2) справедливо для n. И поэтому

$$\text{pair} \rightarrow\rightarrow k^{n+1} \equiv [s \circ a_n \times \circ [a_n, b_n]] \text{ согласно I.1 и I.5.} \quad (6)$$

Для перехода от (5) к (6) нужно проверить, что когда a_n или b_n порождает $\perp$ в (5), то же самое имеет место в (6). Далее,

$$s \circ a_n \equiv s^{n+1} \circ 1 \equiv a_{n+1} \quad (7)$$

$$\times \circ [a_n, b_n] \equiv /\times \circ [s^n \circ 1, \ldots, s \circ 1, 1, 2] \equiv b_{n+1} \text{ согласно I.3} \quad (8)$$

Сочетание (6), (7) и (8) дает

$$\text{pair} \rightarrow\rightarrow k^{n+1} \equiv [a_{n+1}, b_{n+1}]. \quad (9)$$

Итак, отношение (2) справедливо для $n=1$, а также выполняется при $n+1$, если оно верно при n; поэтому по индукции оно справедливо при любом $n \geqslant 1$. Теперь (2) дает для пар

$$\text{defined} \circ k^n \rightarrow\rightarrow p \circ k^n \equiv \text{eq}\,0 \circ 1 \circ [a_n, b_n] \equiv \text{eq}\,0 \circ a_n \equiv \text{eq}\,0 \circ s^n \circ 1 \quad (10)$$

$$\text{defined} \circ k^n \rightarrow\rightarrow g \circ k^n \equiv 2 \circ [a_n, b_n] \equiv /\times \circ [s^{n-1} \circ 1, \ldots, s \circ 1, 1, 2] \quad (11)$$

(в обоих случаях используется I.5). Далее из (1) следует, что $f : \langle x, 1 \rangle$ определено тогда и только тогда, когда имеется такое значение n, для которого $p \circ k^i : \langle x, 1 \rangle = F$ для всех $i < n$ и $p \circ k^n : \langle x, 1 \rangle = T$, т. е., согласно (10), $\text{eq}\,0 \circ s^n : x = T$, т. е. $x = n$; и $g \circ k^n : \langle x, 1 \rangle$ определено; в таком случае, согласно (11),

$$f : \langle x, 1 \rangle = /\times : \langle 1, 2, \ldots, x-1, x, 1 \rangle = n! \quad \square$$

а именно это нам требуется доказать.

12.6.2. Пример: доказательство эквивалентности двух итеративных программ. В этом примере мы хотим показать, что две итеративно описанные программы f и g реализуют одну и ту же функцию. Пусть f — это решение для

$$f \equiv p \circ 1 \rightarrow 2;\ h \circ f \circ [k \circ 1, 2]. \quad (1)$$

Пусть g — решение для

$$g \equiv p \circ 1 \rightarrow 2;\ g \circ [k \circ 1, h \circ 2]. \quad (2)$$

Тогда по теореме об итерации

$f \equiv p_0 \to q_0; \ldots; p_n \to q_n; \ldots$ (3)

$g \equiv p_0' \to q_0'; \ldots; p_n' \to q_n'; \ldots$ (4)

где (при $r^0 \equiv \mathrm{id}$ для любого значения r) для $n=0, 1, \ldots$

$p_n \equiv p \circ 1 \circ [k \circ 1, 2]^n \equiv p \circ 1 \circ [k^n \circ 1, 2]$ по I.5.1 (5)

$q_n \equiv h^n \circ 2 \circ [k \circ 1, 2]^n \equiv h^n \circ 2 \circ [k^n \circ 1, 2]$ по I.5.1 (6)

$p_n' \equiv p \circ 1,\ h \circ [k \circ 1, 2]^n \equiv p \circ 1 \circ [k^n \circ 1,\ h^n \circ 2]$ по I.5.1 (7)

$q_n' \equiv 2 \circ [k \circ 1,\ h \circ 2]^n \equiv 2 \circ [k^n \circ 1,\ h^n \circ 2]$ по I.5.1 (8)

Теперь, воспользовавшись I.5, получаем из сказанного

$\mathrm{defined} \circ 2 \to \to p_n \equiv p \circ k^n \circ 1$ (9)

$\mathrm{defined} \circ h^n \circ 2 \to \to p_n' \equiv p \circ k^n \circ 1$ (10)

$\mathrm{defined} \circ k^n \circ 1 \to \to q_n \equiv q_n' \equiv h^n \circ 2$ (11)

Итак,

$\mathrm{defined} \circ h^n \circ 2 \to \to \mathrm{defined} \circ 2 \equiv \bar{T}$ (12)

$\mathrm{defined} \circ h^n \circ 2 \to \to p_n \equiv p_n'$ (13)

и

$f = p_0 \to q_0; \ldots; p_n \to h^n \circ 2; \ldots$ (14)

$g = p_0' \to q_0'; \ldots; p_n' \to h^n \circ 2; \ldots$ (15)

поскольку из p_n и p_n' следует нужное ограничение для $q_n = q_n' = h^n \circ 2$.

Теперь предположим, что существует такое выражение x, при котором $f \circ x \neq g : x$. Тогда найдется значение n, такое, что $p_i' : x = p_i' : x = F$ для $i < n$ и $p_n : x \neq p_n' : x$. Согласно (12) и (13), такое может случиться, только если $h^n \circ 2 : x = \bot$. Но так как h сохраняет $\bot$, то $h^m \circ 2 : x = \bot$ для всех $m \geqslant n$. Поэтому $f : x = g : x = \bot$ в силу (14) и (15). Это противоречит предположению, что существует выражение x, при котором $f : x \neq g : x$. Поэтому $f \equiv g$.

Этот пример (принадлежащий Дж. Моррису) более изящно рассматривается в [16]. Однако есть основания считать, что наша интерпретация более конструктивна, более автоматически приводит к постановке ключевых вопросов и обеспечивает лучшее понимание поведения обеих функций.

12.7. НЕЛИНЕЙНЫЕ УРАВНЕНИЯ

Прежние примеры касались «линейных» уравнений (в которых «неизвестная» функция не имеет аргумента, зависящего от нее самой). Остается открытым вопрос о существовании простых разложений, которые решали бы квадратные уравнения и уравнения более высокого порядка.

В предыдущих примерах рассматривались решения для $f \equiv \equiv E(f)$, где выражение E линейно разложимо. В следующем примере фигурирует $E(f)$, которое квадратично и разложимо (но не линейно разложимо).

12.7.1. Пример: доказательство идемпотентности [16]. Пусть f — это решение для

$$f \equiv E(f) \equiv p \to \mathrm{id};\ f_2 \circ h. \tag{1}$$

Мы хотим доказать, что $f \equiv f^2$. Проверяем, что выражение E разложимо (разд. 12.4.1) при следующих аппроксимирующих функциях:

$$f_0 \equiv \bar{\perp} \tag{2а}$$

$$f_n \equiv p \to \mathrm{id};\ \ldots;\ p \circ h^{n-1} \to h^{n-1};\ \bar{\perp} \text{ при } n > 0. \tag{2б}$$

Заметим сначала, что $p \to\!\to f_n \equiv \mathrm{id}$ и поэтому

$$p \circ h^1 \to\!\to f_n \circ h^1 \equiv h^1. \tag{3}$$

Теперь $$E(f) \equiv p \to \mathrm{id};\ \bar{\perp}^2 \circ h \equiv f_1 \tag{4}$$

и

$E(f_n)$

$$\begin{aligned}
&\equiv p \to \mathrm{id};\ f_n \circ (p \to \mathrm{id};\ \ldots;\ p \circ h^{n-1} \to h^{n-1};\ \perp) \circ h \\
&\equiv p \to \mathrm{id};\ f_n \circ (p \circ h \to h;\ \ldots;\ p \circ h^n \to h^n;\ \bar{\perp} \circ h) \\
&\equiv p \to \mathrm{id};\ p \circ h \to f_n \circ h;\ \ldots;\ p \circ h^n \to f_n \circ h^n,\ f_n \circ \perp \\
&\equiv p \to \mathrm{id};\ p \circ h \to h;\ \ldots;\ p \circ h^n \to h^n;\ \perp \quad \text{by (3)} \\
&\equiv f_{n+1}
\end{aligned} \tag{5}$$

Итак, выражение E разложимо согласно (4) и (5); поэтому в силу (2) и разд. 12.4.1 (E4)

$$f \equiv p \to \mathrm{id};\ \ldots;\ p \circ h^n \to h^n,\ \ldots \tag{6}$$

Но, по теореме об итерации, (6) дает

$$f \equiv p \to \mathrm{id};\ f \circ h. \tag{7}$$

Теперь, если $p : x = T$, то $f : x = x = f^2 : x$ в силу (1). Если $p : x = \equiv F$, то

$$\begin{aligned}
f : x &= f^2 \circ h : x && \text{по (1)} \\
&= f : (f \circ h : x) = f : (f : x) && \text{по (7)} \\
&= f^2 : x.
\end{aligned}$$

Если $p : x$ не является ни T, ни F, то $f : x = \perp = f^2 : x$. Таким образом, $f \equiv f^2$.

12.8. ОСНОВАНИЯ ДЛЯ АЛГЕБРЫ ПРОГРАММ

В этом разделе мы преследуем цель установить справедливость результатов, сформулированных в разд. 12.4. Последующие разделы не зависят от этого материала, и поэтому читатели могут при желании пропустить его. Мы пользуемся стандартными понятиями и результатами из [16], но для объектов и функций применяется нотация, принятая в этой статье.

В качестве области определения (и области значений) для всех функций выбирается множество O объектов (включая $\perp$) данной системы ФП. Будем обозначать через F множество функций, а через **F** — множество функциональных форм такой системы ФП. Обозначаем как E(f) любое функциональное выражение, включающее функциональные формы, примитивные и выведенные функции, а также символ функции f, и будем относиться к E как к функционалу, который отображает функцию f на соответствующую функцию E (f). Предполагаем, что все $f \in$ F сохраняют $\perp$ и что все функциональные формы из **F** соответствуют непрерывным функционалам по каждому переменному (например, [f, g] непрерывна как по f, так и по g). (Все примитивные функции представленной ранее системы ФП сохраняют $\perp$, а все ее функциональные формы непрерывны.)

Определение. Пусть E(f) — это функциональное выражение. Пусть

$f_0 \equiv \overline{\perp}$

$f_{i+1} \equiv p_0 \to q_0; \ldots; p_i \to q_i; \overline{\perp}$ при $i = 0, 1, \ldots$

где $p_i, q_i \in F$. Пусть E обладает тем свойством, что

$E(f_i) \equiv f_{i+1}$ при $i = 0, 1, \ldots$.

Тогда выражение E называется *разложимым с аппроксимирующими функциями* f_i. Мы пишем

$f \equiv p_0 \to q_0; \ldots; p_n \to q_n; \ldots$

чтобы указать, что $f \equiv \lim_i \{f_i\}$, где f_i имеет приведенный выше вид. Правую часть мы называем *бесконечным разложением* для f. Будем считать, что $f : x$ определено тогда и только тогда, когда существует значение $n \geqslant 0$, такое, что (а) $p_1 : x = \mathbf{F}$ для всех $i < n$ и (б) $p_n : x = T$ и (в) $q_n : x$ определено; в таком случае $f : x = q_n : x$.

Теорема о разложении. *Пусть выражение* E(f) *разложимо с указанными выше аппроксимирующим функциями. Пусть* f — *наименьшая функция, удовлетворяющая*

$f = E(f)$.

Тогда

$f \equiv p_0 \to q_0; \ldots; p_n \to q_n; \ldots .$

Доказательство. Поскольку E — это композиция непрерывных функционалов (из **F**), включающих только монотонные функции (сохраняющие $\bot$ функции из F) в качестве константных членов, то функция E непрерывна [16]. Поэтому ее наименьшая фиксированная точка f имеет вид $\lim_i\{E^i(\overline{\bot})\} \equiv \equiv \lim_i\{f_i\}$, а по определению это и есть указанное выше бесконечное разложение для f.

Определение. Пусть $E(f)$ — функциональное выражение, удовлетворяющее

$E(h) \equiv p_0 \to q_0;\ E_1(h)$ при всех $h \in F$, (LE1)

причем существуют $p_i \in F$ и $q_i \in F$, такие, что

$E_1(p_i \to q_i;\ h) \equiv p_{i+1} \to q_{i+1};\ E_1(h)$ при всех $h \in F$ и $i = 0, 1, \ldots$ (LE2)

и

$E_1(\overline{\bot}) \equiv \overline{\bot}.$ (LE3)

Тогда выражение E называется *линейно разложимым* по этим p_i и q_i.

Теорема о линейном разложении. *Пусть выражение* E *линейно разложимо по* p_i *и* q_i, $i = 0, 1, \ldots$ *Тогда выражение* E *разложимо с аппроксимирующими функциями*

$f_0 \equiv \overline{\bot}$ (1)

$f_{i+1} \equiv p_0 \to q_0; \ldots; p_i \to q_i;\ \overline{\bot}.$ (2)

Доказательство. Мы хотим показать, что $E(f_i) \equiv f_{i+1}$ для любого значения $i \geqslant 0$. Теперь

$E(f_0) \equiv p_0 \to q_0;\ E_1(\overline{\bot}) \equiv p_0 \to q_0;\ \bot \equiv f_1$ по (LE1)(LE3)(1). (3)

Пусть фиксировано значение $i > 0$ и пусть

$f_i \equiv p_0 \to q_0;\ w_1$ (4а)

$w_1 \equiv p_1 \to q_1;\ w_2$ и т. д. (4б)

$w_{i-1} \equiv p_{i-1} \to q_{i-1};\ \overline{\bot}.$ (4-)

Тогда для этого решения $i > 0$

$E(f_i) \equiv p_0 \to q_0;\ E_1(f_i)$ по (LE1) (LE1)

$E_1(f_i) \equiv p_1 \to q_1;\ E_1(w_1)$ по (LE2) и (4а)

$E_1(w_1) \equiv p_2 \to q_2;\ E_1(w_2)$ по (LE2) и (4б) и т. д.

$E_1(w_{i-1}) \equiv p_i \rightarrow q_i$; $E_1(\overline{\perp})$ по (LE2) и (4-)
$\equiv p_i \rightarrow q_i$; $\perp$ по (LE3)

Сочетание этих отношений дает

$E(f_i) \equiv f_{i+1}$ для произвольного $i>0$ по (2). (5)

В силу (3) отношение (5) также справедливо при $i \geqslant 0$, поэтому оно верно для всех значений $i \geqslant 0$. Следовательно, выражение E разложимо и обладает требуемыми аппроксимирующими функциями.

Следствие. *Если выражение* E *линейно разложимо по* p_i *и* q_i *при* $i=0, 1, \ldots$ *и* f *— наименьшая функция, удовлетворяющая*

$f \equiv E(f)$ (LE4)

то

$f \equiv p_0 \rightarrow q_0; \ldots; p_n \rightarrow q_n; \ldots$. (LE5)

12.9. АЛГЕБРА ПРОГРАММ ДЛЯ ЛЯМБДА-ИСЧИСЛЕНИЯ И ДЛЯ КОМБИНАТОРОВ

Поскольку лямба-исчисление Чёрча [5] и система комбинаторов, разработанная Шёнфинкелем и Карри [6], являются в первую очередь математическими системами для представления понятия применения функций и поскольку они более мощны, чем системы ФП, то естественно заняться исследованием того, как будет выглядеть основанная на этих системах алгебра программ. В лямбда-исчислении и комбинаторах эквиваленты для композиции $f \circ g$ из ФП имеют вид

$\lambda fgx.\ (f(gx)) \equiv B$

где B — простой комбинатор, определенный Карри. В системах Чёрча или Карри отсутствуют подходящие непосредственные эквиваленты для объекта $\langle x, y \rangle$ из ФП, однако, согласно Ландину [14] и Бёрджу [4], можно воспользоваться примитивными функциями префикс, голова, хвост, нуль и атом, чтобы ввести понятие списковых структур, которые соответствуют последовательностям из ФП. Тогда применив понятие из ФП для списка, можно сделать эквивалентом конструкции ФП выражение $\lambda fgx.\ \langle fx, gx \rangle$ из лямбда-исчисления. Эквивалентом в комбинаторах является выражение, включающее префикс, нулевой список и два или более основных комбинатора. Оно столь сложно, что я не пытаюсь привести его здесь.

При использовании лямбда-исчисления или комбинаторных выражений для функциональных форм $f \circ g$ и $[f, g]$ для пред-

ставления закона I.1 из алгебры ФП, $[f, g]\circ h \equiv [f\circ h, g\circ h]$, результатом является настолько сложное выражение, что смысл закона затемняется. Единственный способ сделать этот смысл ясным в любой системе состоит в том, чтобы вывести наименования для двух функционалов: композиция $\equiv$B и конструкция $\equiv$A, так что $Bfg \equiv f\circ g$ и $Afg \equiv [f,g]$.
Тогда I.1 принимает вид

$$B(Afg)h \equiv A(Bfh)(Bgh)$$

который все еще не столь прозрачен, как закон из ФП.

Суть сказанного состоит в том, что если некто хочет сформулировать ясные законы в системах Чёрча или Карри наподобие соответствующих законов в алгебре ФП, то он сталкивается с необходимостью выбрать определенные функционалы (например, композицию и конструкцию) в качестве основных операций алгебры и либо присвоить им короткие имена, либо, что предпочтительнее, представить их некоторой специальной нотацией, как в ФП. Если сделать это и обеспечить примитивы, объекты, списки и т. д., результатом будет ФП-подобная система, в которой не появляются обычные лямбда-выражения или комбинаторы. Даже тогда эти версии Чёрча или Карри для системы ФП в силу меньшей ограниченности представляют некоторые проблемы, которые не возникают в системах ФП:

(а) Версии Чёрча и Карри предоставляют функции многих типов и позволяют описывать функции, которые не существуют в системах ФП. Так, функция Bf не имеет эквивалента в системах ФП. Эта дополнительная мощь влечет за собой проблемы совместимости типов. Например, для функции $f\circ g$ включается ли диапазон значений g в область определения f? В системах ФП все функции обладают одними и теми же областями определения и значений.

(б) Семантика лямбда-исчисления Чёрча зависит от правил подстановок, которые просто формулируются, но следствия которых с большим трудом поддаются осознанию. Далеко не все понимают истинную сложность этих правил, но ее очевидным свидетельством является преуспевание логиков, публиковавших «доказательства» теоремы Чёрча — Россера, в которых упускались из виду те или иные из этих сложностей. (Теорема Чёрча — Россера или доказательство Скотта существования модели [22] требуются для того, чтобы показать, что у лямбда-исчисления имеется непротиворечивая семантика.) Описание чистого Лиспа длительное время содержало связанную с этим ошибку (проблема «funarg»). Аналогичные проблемы возникают и в системе Карри.

Напротив, формальная (ФФП) версия систем ФП (описываемая в следующем разделе) не содержит переменных и

включает только одно элементарное правило подстановки (функция вместо ее имени), и для нее существование непротиворечивой семантики может быть доказано относительно просто в духе рассуждений, развитых Д. Скоттом, а также Манной и др. [16]. Такое доказательство приведено в [18].

12.10. ЗАМЕЧАНИЯ

Намеченная выше алгебра программ потребует большой работы для распространения ее на более обширные классы или уравнения и для обобщения ее законов и теорем за рамки приведенных здесь элементарных случаев. Было бы интересно исследовать алгебру для ФП-подобной системы, в которой конструктор последовательности не сохраняет $\perp$ (закон 1.5 становится более сильным, но теряется IV.1). Другие интересные проблемы состоят в том, чтобы (а) найти правила обеспечения однозначности разложений, задав канонические формы для функций; (б) найти алгоритмы разложения и анализа поведения функций для различных классов аргументов и (в) исследовать способы использования законов и теорем алгебры в качестве основных правил либо для схемы формального, умозрительного «ленивого вычисления» [9, 10], либо для схемы, работающей в процессе выполнения алгоритма. В таких схемах, например, закон $1\circ[f, g]\leqslant f$ служил бы для того, чтобы избежать вычисления $g:x$.

13. ФОРМАЛЬНЫЕ СИСТЕМЫ ДЛЯ ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ (СИСТЕМЫ ФФП)

13.1. ВВЕДЕНИЕ

Как мы уже видели, система ФП включает множество функций, которое зависит от ее множества примитивных функций, ее множества функциональных форм и ее множества определений. В частности, ее множество функциональных форм фиксировано раз навсегда, и это множество решающим образом определяет мощь системы. Например, если ее множество функциональных форм пусто, то ее множество собственно функций в точности совпадает с множеством примитивных функций. В системах ФФП можно создавать новые функциональные формы. Функциональные формы представляются последовательностями объектов; первый элемент последовательности определяет, какую форму она представляет, тогда как остальные элементы являются параметрами формы.

Возможность описывать в системах ФФП новые функциональные формы является одним из следствий принципиального

различия между этими системами и системами ФП: в системах ФФП объекты служат для «представления» функций систематическим образом. В других отношениях ФФП почти не отличаются от ФП. Они похожи на языки сведения (Red) из ранней публикации [2], но проще этих языков.

Сначала мы приведем простой синтаксис систем ФФП, затем неформально обсудим их семантику, рассмотрев примеры, и, наконец, введем их формальную семантику.

13.2. СИНТАКСИС

Мы описываем множество O объектов и множество E выражений системы ФФП. Они зависят от выбора некоего множества A *атомов*, которое мы принимаем как заданное. Мы предполагаем, что множеству A принадлежат T (истина), F (ложь), $\varnothing$ (пустая последовательность), а также «числа» различных видов и т. д.

(1) Основа $\perp$ является *объектом*, но не атомом.

(2) Всякий атом является *объектом*.

(3) Всякий объект является *выражением*.

(4) Если $x_1, \ldots, x_n$ — это объекты (выражения), то $\langle x_1, \ldots, x_n\rangle$ является *объектом* (соответственно *выражением*), называемым *последовательностью* (*длины* n) при $n \geqslant 1$. Объект (выражение) x_i при $1 \leqslant i \leqslant n$ является *элементом* последовательности $\langle x_1, \ldots, x_i, \ldots, x_n\rangle$ ($\varnothing$ одновременно является и последовательностью, и атомом; длина этой последовательности равна 0).

(5) Если x и y — это выражения, то $(x:y)$ является *выражением*, называемым *применением*. Здесь x — оператор, а y — *операнд*. И x, и y представляют собой *элементы* выражения.

(6) Если $x=\langle x_1, \ldots, x_n\rangle$ и если один из элементов x есть $\perp$, то $x=\perp$. Иначе говоря, $\langle \ldots, \perp, \ldots\rangle = \perp$.

(7) Все объекты и выражения формируются использованием приведенных выше правил конечное число раз.

Подвыражением выражения x является либо само x, либо подвыражение некоего элемента из x. Объект ФФП представляет собой выражение, не содержащее применения в качестве своего подвыражения. Если задано одно и то же множество атомов, то объекты ФФП и ФП одинаковы.

13.3. НЕФОРМАЛЬНЫЕ ЗАМЕЧАНИЯ О СЕМАНТИКЕ ФФП

13.3.1. Значение выражения; семантическая функция μ. Всякое выражение e из ФФП имеет *значение* μe, которое всегда является объектом; μe отыскивается повторяемой заменой всякого самого внутреннего применения в e на его значение. Если этот процесс не завершается, то значение e есть $\perp$. Значением

самого внутреннего применения $(x:y)$ (поскольку оно самое внутреннее, то x и y должны быть объектами) является результат применения к y функции, *представляемой* x, точно так же, как в системах ФП, с той разницей, что в системах ФФП функции представляются объектами, а не функциональными выражениями, причем атомы (а не символы функций) представляют примитивные и определяемые функции, а последовательности представляют функции ФП, обозначаемые функциональными формами.

Связь между объектами и представляемыми ими функциями задается *функцией представления* ρ системы ФФП. (И ρ, и μ принадлежат описанию системы, но не самой системе.) Итак, если атом *NULL* представляет функцию null системы ФП, то $\rho NULL = \text{null}$ и значение $(NULL:A)$ есть $\mu(NULL:A) = (\rho NULL):A = \text{null}:A = F$.

Здесь, как и выше, мы используем двоеточие в двух смыслах. Когда оно находится между двумя объектами, как в $(NULL:A)$, оно идентифицирует применение ФФП, которое обозначает только себя; когда же оно находится между *функцией* и объектом, как в $(\rho NULL):A$ или в $\text{null}:A$, оно идентифицирует ФП-подобное применение, которое обозначает *результат* применения функции к объекту.

Тот факт, что операторы ФФП являются объектами, делает возможным существование функции apply, которая не имеет смысла в системах ФП:

$$\text{apply}:\langle x, y\rangle = (x:y)$$

Результат $\text{apply}:\langle x, y\rangle$, а именно $(x:y)$, не имеет смысла в системах ФП, причем это верно на двух уровнях. Во-первых, сочетание $(x:y)$ само по себе не является объектом; оно иллюстрирует другое различие между системами ФП и ФФП: некоторые функции из ФФП, например apply, отображают объекты на выражения, а не непосредственно на объекты (в отличие от функций из ФП). Однако *значение* $\text{apply}:\langle x, y\rangle$ представляет собой объект (см. ниже). Во-вторых, $\langle x:y\rangle$ не может быть даже промежуточным результатом в системе ФП; это сочетание не имеет смысла в системах ФП, потому что x — это объект, а не функция, а системы ФП не ассоциируют функции с объектами. Теперь если *APPLY* представляет apply, то значение $(APPLY:\langle NULL, A\rangle)$ равно

$$\begin{aligned}\mu(APPLY:\langle NULL,A\rangle) &= \mu((\rho APPLY):\langle NULL,A\rangle)\\ &= \mu(\text{apply}:\langle NULL,A\rangle)\\ &= \mu(NULL:A) = \mu((\rho NULL):A)\\ &= \mu(\text{null}:A) = \mu F = F.\end{aligned}$$

Последний шаг вытекает из того факта, что всякий объект является своим собственным значением. Поскольку функция значения μ в конечном счете вычисляет все применения, то можно рассматривать apply: $\langle NULL, A\rangle$ как порождение F, несмотря на то что фактический результат есть $(NULL : A)$.

13.3.2. Как объекты представляют функции; функция представления ρ. Как мы уже видели, некоторые атомы (*примитивные* атомы) будут представлять примитивные функции системы. Другие атомы могут представлять определяемые функции точно так же, как символы делают это в системах ФП. Если атом не является ни примитивным, ни определяемым, он представляет $\overline{\perp}$, функцию, которая повсюду есть $\perp$.

Последовательности тоже представляют функции и аналогичны функциональным формам в ФП. Функция, представляемая последовательностью, задается (рекурсивно) следующим правилом.

Правило метакомпозиции

$$(\rho\langle x_1, \ldots, x_n\rangle) : y = (\rho x_1) : \langle\langle x_1, \ldots, x_n\rangle, y\rangle,$$

где x и y являются объектами. Здесь ρx_1 определяет, что представляет функциональная форма $\langle x_1, \ldots, x_n\rangle$, а $x_2, \ldots, x_n$ являются параметрами формы (в ФФП сам объект x_1 может также служить параметром). Так, например, пусть **Def** ρ $CONST \equiv \equiv 2\circ 1$; тогда $\langle CONST, x\rangle$ в ФФП представляет функциональную форму x из ФП, потому что, согласно правилу метакомпозиции, если $y \neq \perp$, то

$$(\rho\langle CONST, x\rangle) : y = (\rho CONST) : \langle\langle CONST, x\rangle, y\rangle = 2\circ 1\langle\langle CONST, x\rangle, y\rangle = x.$$

Здесь мы можем видеть, что первый управляющий оператор из последовательности или формы, в данном случае *CONST*, всегда имеет своим операндом после метакомпозиции пару, в которой первый элемент сам является последовательностью, а второй элемент является исходным операндом последовательности, в данном случае y. Управляющий оператор может переупорядочивать и применять заново элементы последовательности и исходный операнд самыми разнообразными способами. Существенный аспект метакомпозиции состоит в том, что она позволяет описывать новые функциональные формы просто путем определения новых функций. Она позволяет также записывать рекурсивные функции без определения.

Рассмотрим еще один пример контролирующей функции для функциональной формы **Def** ρ $CONS \equiv \alpha$ apply$\circ$tl$\circ$distr. Это определение приводит к выражению $\langle CONS, f_1, \ldots, f_n\rangle$ (где f_i — суть объекты), представляющему ту же функцию, что и $[\rho f_1, \ldots, \rho f_n]$. Ниже доказывается это утверждение.

$(\rho\langle CONS, f_1, \ldots, f_n\rangle) : x$
$= (\rho CONS) : \langle\langle CONS, \ldots, f_n\rangle, x\rangle$ в силу метакомпозиции
$=_\alpha$ apply∘tl∘distr : $\langle\langle CONS, f_1, \ldots, f_n\rangle, x\rangle$ по определению $\rho CONS$
$=_\alpha$ apply : $\langle\langle f_1, x\rangle, \ldots, \langle f_n, x\rangle\rangle$ по определению tl, distr и ∘
$=\langle$apply : $\langle f_1, x\rangle, \ldots,$ apply : $\langle f_n, x\rangle\rangle$ по определению α
$=\langle (f_1 : x), \ldots, (f_n : x)\rangle$ по определению apply.

При вычислении последнего выражения семантическая функция μ будет порождать значение каждого применения, давая $\rho f_i : x$ в качестве i-го элемента.

Обычно при описании функции, представляемой последовательностью, мы будем задавать ее общий эффект, а не показывать, как ее управляющий оператор обеспечивает этот эффект. Таким образом, мы будем просто писать

$$(\rho\langle CONS, f_1, \ldots, f_n\rangle) : x = \langle (f_1 : x), \ldots, (f_n : x)\rangle$$

вместо более детальной записи, приведенной выше.

Нам нужен управляющий оператор *COMP* для получения последовательностей, представляющих композицию функциональных форм. Будем считать, что $\rho COMP$ является примитивной функцией, такой, что для всех объектов x

$$(\rho\langle COMP, f_1, \ldots, f_n\rangle) : x = (f_1 : (f_2 : (\ldots : (f_n : x)\ldots))) \quad \text{при } n \geqslant 1.$$

(Я признателен П. Макджонсу за его наблюдение, что обычную композицию можно получить с помощью этой примитивной функции вместо использования двух правил композиции в основной семантике, как было сделано в ранней работе [2].)

Хотя системы ФФП позволяют описывать и исследовать новые функциональные формы, следует ожидать, что большинство программистов ограничились бы использованием фиксированного набора форм (управляющие операторы которых являются примитивными), как в системах ФП, так что можно было бы применять алгебраические законы для этих форм и воспользоваться стилем структурного программирования на основе этих форм.

Помимо использования в определениях функциональных форм, метакомпозиция может служить для непосредственного создания рекурсивных функций без применения определения вида **Def** $f \equiv E(f)$. Например, если $\rho MLAST =$ null∘tl∘2$\rightarrow$1∘2; apply∘[1, tl∘2], то $\rho\langle MLAST\rangle \equiv$ last, где last : $x \equiv x = \langle x_1, \ldots, \ldots, x_n\rangle \rightarrow x_n$; $\perp$. Таким образом, оператор $\langle MLAST\rangle$ работает следующим образом:

$\mu(\langle MLAST\rangle : \langle A, B\rangle)$
$= \mu(\rho MLAST : \langle\langle MLAST\rangle, \langle A, B\rangle\rangle)$ в силу метакомпозиции

$= \mu(\text{apply} \circ [1, \text{tl} \circ 2] : \langle\langle MLAST\rangle, \langle A, B\rangle\rangle)$
$= \mu(\text{apply} : \langle\langle MLAST\rangle, \langle B\rangle\rangle)$
$= \mu(\langle MLAST\rangle : \langle B\rangle)$
$= \mu(\rho MLAST : \langle\langle MLAST\rangle, \langle B\rangle\rangle)$
$= \mu(1 \circ 2\langle\langle MLAST\rangle, \langle B\rangle\rangle)$
$= B.$

13.3.3. Резюме свойств функций ρ и μ. Ранее мы показали, как функция ρ отображает атомы и последовательности на функции и как эти функции отображают объекты на выражения. Фактически функция ρ и все функции ФФП могут быть обобщены таким образом, что становятся определенными для всех выражений. При таких обобщениях свойства функций ρ и μ могут быть подытожены следующим образом:

(1) $\mu \in$ [выражение → объекты].

(2) Если x — это объект, то $\mu x = x$.

(3) Если e является выражением и $e = \langle e_1, \ldots e_n\rangle$, то $\mu e = \langle \mu e_1, \ldots, \mu e_n\rangle$.

(4) $\rho e \in$ [выражения→выражения→выражения]]

(5) Для любого выражения e, $\rho e = \rho(\mu e)$.

(6) Если x — объект, а e — выражение, то $\rho x : e = \rho x : (\mu e)$.

(7) Если x и y — объекты, то $\mu(x : y) = \mu(\rho x : y)$. В словесном выражении это означает, что значение применения $(x : y)$ в системе ФФП отыскивается применением к y функции ρx, представимой с помощью x, а затем нахождением значения выражения-результата (которое *обычно* является объектом и в таком случае представляет собой собственное значение).

13.3.4. Ячейки, доставка и запоминание. В силу ряда причин удобно строить функции, служащие в роли имен. В частности, нам понадобится эта возможность для описания семантики определений в системах ФФП. Чтобы ввести именующие функции, т. е. возможность *доставить* (*fetch*) содержимое ячейки с данным именем из памяти (последовательности ячеек) и для *запоминания* (*store*) ячейки с данным именем и содержимым в такой последовательности, мы вводим объекты, называемые *ячейками* (*cell*), и две новые функциональные формы *fetch* и *store*.

Ячейки. Ячейка представляет собой триплет (CELL, *имя, содержимое*). Мы используем эту форму вместо пары (*имя, содержимое*), так что ячейки можно отличить от обычных пар.

Доставка. Функциональная форма *fetch* использует в качестве своего параметра объект n (обычно n — это атом, служащий в роли имени); это записывается $\uparrow n$ (читается «доставить n»). Описание этой функциональной формы для объектов n и x имеет вид

$\uparrow n : x \equiv x = \varnothing \to \#$; atom : $x \to \bot$; $(1 : x) = \langle CELL, n, c \rangle \to c$; $\uparrow n \circ \mathrm{tl} : x$

где # — это атом «отсутствие». Таким образом, $\uparrow n$ (доставить n) в применении к последовательности дает содержимое первой ячейки из последовательности, имя которой есть n; если не существует ячейки с именем n, то результатом является отсутствие, #. Итак, $\uparrow n$ — это имя функции для имени n. (Мы предполагаем, что $\rho FETCH$ — это примитивная функция, такая, что $\rho\langle FETCH, n\rangle \equiv \uparrow n$. Заметим, что $\uparrow n$ просто пропускает элементы в своем операнде, которые не являются ячейками.)

Запоминание, помещение на стек, снятие со стека, чистка. Подобно доставке, функциональная форма *store* берет в качестве своего параметра объект n; она записывается $\downarrow n$ («запомнить n»). При применении к паре (x, y), где y — это последовательность, $\downarrow n$ удаляет из y первую ячейку с именем n, если такая есть, а затем создает новую ячейку с именем n и содержимым x и присоединяет ее к y. Прежде чем определять $\downarrow n$ (запомнить n), мы специфицируем четыре вспомогательные функциональные формы. (Они могут быть использованы в сочетании с fetch n и store n для получения множественных именованных стеков LIFO (последним пришел — первым ушел) в рамках последовательности памяти.) Две из этих вспомогательных форм специфицируются рекурсивными функциональными уравнениями; в каждой из них в роли параметра выступает объект n. Ниже cellname — это имя ячейки, push — помещение n на стек, pop n — снятие со стека в n, purge n — чистка n.

$$(\mathrm{cellname}\ n) \equiv \mathrm{atom} \to \overline{F};\ \mathrm{eq} \circ [\mathrm{length}, \overline{3}] \to \mathrm{eq} \circ [[\overline{\mathrm{CELL}}, n], [1, 2]];\ \overline{F}$$

$$(\mathrm{push}\ n) \equiv \mathrm{pair} \to \mathrm{apndl} \circ [[\overline{\mathrm{CELL}}, \overline{n}, 1], 2];\ \overline{\bot}$$

$$(\mathrm{pop}\ n) \equiv \mathrm{null} \to \overline{\varnothing};\ (\mathrm{cellname}\ n) \circ 1 \to \mathrm{tl};\ \mathrm{apndl} \circ [1, (\mathrm{pop}\ n) \circ \mathrm{tl}]$$

$$(\mathrm{purge}\ n) \equiv \mathrm{null} \to \overline{\varnothing};\ (\mathrm{cellname}\ n) \circ \to (\mathrm{purge}\ n) \circ \mathrm{tl};\ \mathrm{apndl} \circ [1, (\mathrm{purge}\ n) \circ \mathrm{tl}]$$

$$\downarrow n \equiv \mathrm{pair}\,(\mathrm{push}\ n) \circ [1, (\mathrm{pop}\ n) \circ 2];\ \overline{\bot}$$

Приведенные выше функциональные формы работают следующим образом. Для $x \neq \bot$ имеем $(\mathrm{cellname}\ n) : x$ равно T, если x — это ячейка с именем n, или равно F в противном случае; $(\mathrm{pop}\ n) : y$ удаляет первую ячейку с именем n из последовательности y; $(\mathrm{purge}\ n) : y$ удаляет из y все ячейки с именем n; $(\mathrm{push}\ n) : \rangle x, y\rangle$ помещает ячейки с именем n и содержимым x в начало последовательности y; $\downarrow n : \langle x, y\rangle$ равно $(\mathrm{push}\ n) : \langle x, (\mathrm{pop}\ n) : y\rangle$.

(Таким образом, $(\mathrm{push}\ n) : \langle x, y\rangle$ помещает x на вершину

«стека» с именем n в y'; x можно прочесть с помощью $\uparrow n : y' = x$ и можно удалить с помощью $(\mathrm{pop}\, n) : y'$. Итак, $\uparrow n \circ (\mathrm{pop}\, n) : y'$ — это элемент под x в стеке n при условии, что в y' имеется более чем одна ячейка с именем n.)

13.3.5. Определения в системах ФФП. Семантика в системе ФФП зависит от фиксированного множества определений D (последовательности ячеек) точно так же, как система ФП зависит от ее неформально заданного множества определений. Таким образом, семантическая функция μ зависит от D; изменение D дает новую функцию μ', которая отражает измененные определения. Мы представляли D как объект, потому что в системах ФСПС (разд. 14) мы захотим преобразовывать множество D путем применения к нему функций и доставлять из него данные, кроме его использования в качестве источника определений функций в семантике ФФП.

Если $\langle CELL, n, c\rangle$ — это первая ячейка с именем n в последовательности D (и n — это атом), то она действует как описание **Def** $n = \rho c$ в ФП, т. е. значение $(n : x)$ такое же, как значение $\rho c : x$. Например, если $\langle CELL, CONST, \langle COMP, 2, 1\rangle\rangle$ — это первая ячейка в D с именем $CONST$, то она действует так же, как **Def** $CONST \equiv 2 \circ 1$, и при этом множестве D система ФФП нашла бы $\mu(CONST : \langle\langle x, y\rangle, z\rangle) = y$, и следовательно, $\mu(\langle CONST, A\rangle : B) = A$.

В общем случае в системе ФФП с описаниями D значение применения формы $(atom : x)$ зависит от D; если $\uparrow atom : \mathrm{D} \neq \#$ (т. е. *атом* описан в D), то его значение есть $\mu(c : x)$, где $c = \uparrow atom : \mathrm{D}$, т. е. содержимое первой ячейки в D с именем *atom*. Если $\uparrow atom\, \mathrm{D} = \#$, то *atom* не описан в D, и либо *atom* является примитивом, т. е. система знает, как вычислять $\rho\, atom : x$ и $\mu(atom : x) = \mu(\rho\, atom : x)$, либо в противном случае $\mu(atom : x) = \bot$.

13.4. ФОРМАЛЬНЫЕ СЕМАНТИКИ ДЛЯ СИСТЕМ ФФП

Мы предполагаем, что множество A атомов, множество D определений, множество $P \subset A$ примитивных атомов и представляемых ими примитивных функций уже выбраны. Предполагаем также, что ρa — это примитивная функция, представляемая a, если a принадлежит множеству P, и что $\rho a = \bot$, если a принадлежит Q, множеству атомов в A—P, которые не описаны в D. Хотя функция ρ определена для всех выражений (см. 13.3.3), формальная семантика использует ее определение только в множествах P и Q. Те функции, которые ρ присваивает другим выражениям x, неявно определены и применяются для вычисления $\mu(x : y)$ по следующим семантическим правилам:

Упомянутый выше выбор A, D и P и соответствующих примитивных функций определяет объекты, выражения и семантическую функцию μ_D для системы ФФП. (Мы считаем множество D фиксированным и пишем μ вместо μ_D.) Предполагаем, что D — это последовательность и что функция $\uparrow y : D$ может быть вычислена (функция $\uparrow$ задана в разд. 13.3.4) для любого атома y. При этих предположениях мы описываем μ как наименьшую фиксированную точку функционала τ, где функция $\tau\mu$ для любой функции μ описывается следующим образом (для всяких выражений x, x_i, y, y_i, z и w):

$$
\begin{aligned}
(\tau\mu)x \equiv\ & x \in A \rightarrow x;\\
& x = \langle x_1, \ldots, x_n\rangle \rightarrow \langle \mu x_1, \ldots, \mu x_n\rangle;\\
& x = (y:z) \rightarrow\\
& \quad (y \in A \ \&\ (\uparrow y:D) = \# \rightarrow \mu((\rho y)(\mu z));\\
& \quad y \in A \ \&\ (\uparrow y:D) = w \rightarrow \mu(w:z);\\
& \quad y = \langle y_1, \ldots, y_n\rangle \rightarrow \mu(y_1 : \langle y,z\rangle);\ \mu(\mu y:z));\ \bot
\end{aligned}
$$

Приведенное выше описание μ разлагает оператор применения посредством определений и метакомпозиции, прежде чем вычислить операнд. Предполагается, что в указанном описании $\tau\mu$ предикаты виды "$x \in A$" сохраняют $\bot$ (например, "$\bot \in A$" имеет значение $\bot$) и что само условное выражение тоже сохраняет $\bot$. Таким образом, $(\tau\mu)\bot \equiv \bot$ и $(\tau\mu)(\bot : z) \equiv \bot$. Этим завершается рассмотрение семантики систем ФФП.

14. ФУНКЦИОНАЛЬНЫЕ СИСТЕМЫ ПЕРЕХОДОВ СОСТОЯНИЙ (СИСТЕМЫ ФСПС)

14.1. ВВЕДЕНИЕ

В этом разделе дается обзор класса систем, упомянутых ранее как альтернативы для систем фон Неймана. Необходимо снова подчеркнуть, что эти функциональные системы перехода состояний выдвигаются на передний план не как практические системы программирования в их современной форме, а как примеры класса, в котором функциональный стиль программирования становится доступным в исторически чувствительных системах, не являющихся системами фон Неймана. Эти системы слабо связаны с состояниями и зависят от лежащей в основе функциональной системы и в отношении их языка программирования, и в отношении описания их переходов состояний. Основная функциональная система описываемой ниже системы ФСПС представляет собой систему ФФП, но могли бы использоваться и другие функциональные системы.

Для понимания причин, мотивирующих выбор структуры систем ФСПС, полезно сначала вспомнить основную структуру одной из систем фон Неймана, а именно Алгола, рассмотреть ее ограничения и сравнить ее со структурой систем ФСПС. После такого обзора описывается минимальная система ФСПС. Представлена небольшая, организованная по принципу сверху вниз, с самозащитой, системная программа для поддержания файлов и организации прохождения пользовательских программ, указываются пути ее реализации в системе ФСПС и выполнения примера программы пользователя. В системной программе используются «именные функции» вместо обычных имен, аналогично может поступать и пользователь. Раздел заключается подразделами, в которых обсуждаются варианты системы ФСПС, их общие свойства и системы именования.

14.2. СРАВНЕНИЕ СТРУКТУРЫ АЛГОЛА СО СТРУКТУРОЙ СИСТЕМ ФСПС

Программа на Алголе является последовательностью операторов, каждый из которых представляет преобразование состояния Алгола, т. е. сложного сочетания информации о состояниях различных стеков, указателей, переменных, отображающих идентификаторы на значения, и т. д. Всякий оператор связывается с этим постоянно изменяющимся состоянием посредством изощренных протоколов, каждый из которых зависит от самого оператора и даже от его составных частей (например, протокол, ассоциирующийся с переменной x, зависит от ее появления в левой или правой части присваивания, в описании, в виде параметра и т. д.).

Дело обстоит так, будто состояние Алгола представляет собой сложную «память», которая связывается с программой на Алголе через огромный «кабель», состоящий из многих специализированных проводов. Сложные протоколы связей этого кабеля фиксированы и включают в себя протоколы для каждого типа операторов. «Значение» программы на Алголе должно выражаться через общий эффект огромного количества актов коммуникаций с состоянием посредством кабеля и его протоколов (а также через средства идентификации выходных данных и включения входных данных в состояние). По сравнению с этим массивным кабелем, связывающим состояние и память Алгола с программой на Алголе, тот кабель, который представляет «бутылочное горлышко» компьютера фон Неймана, является простым, изящным понятием.

Таким образом, операторы Алгола не являются выражениями, представляющими функции перехода из одного состояния в другое и строящимися из более простых функций переходов

состояний с использованием упорядоченных комбинационных форм. Напротив, это сложные *сообщения* с зависящими от контекста частями, которые «обгрызаются» состоянием. Каждая часть передает информацию в состояние и из состояния через кабель по его собственным протоколам. Не предусматривается средств для применения общих функций ко *всему* состоянию и тем самым для внесения в него больших изменений. Возможность больших, мощных преобразований состояния S применением функций $S \rightarrow f : S$ на самом деле невообразима в контексте кабеля и протоколов фон Неймана: ведь не было бы уверенности, что новое состояние $f : S$ окажется согласованным с кабелем и его фиксированными протоколами, если только f не ограничивается мелкими изменениями, которые прежде всего должны допускаться кабелем.

Нам желательна вычислительная система, семантика которой не зависит от множества причудливых протоколов для связи с состоянием, и мы хотим уметь вносить в состояния большие преобразования путем применения общих функций. Системы ФСПС обеспечивают один из способов достижения этих целей. В их семантике имеются два протокола для получения информации из состояния: (1) получение из него определения функций, которую нужно применить, и (2) получение самого состояния в целом. Имеется один протокол для изменения состояния: вычисление нового состояния применением функции. За исключением актов коммуникации с состоянием, семантика ФСПС является функциональной (т. е. соответствует системе ФФП). Она не зависит от изменений состояний, потому что в течение вычисления состояние совсем не изменяется. Наоборот, результат вычислений образует входные данные и новое состояние. Структура состояния системы ФСПС несколько ограничена одним из ее протоколов: должна иметься возможность идентифицировать определение (т. е. ячейку) в этом состоянии. Его структура имеет вид последовательности и гораздо более проста, чем структура состояния в Алголе.

Итак, структура систем ФСПС свободна от сложности и ограничений состояния фон Неймана (с его протоколами связи) и достигает большей мощи и свободы совсем другим и более простым способом.

14.3. СТРУКТУРА СИСТЕМЫ ФСПС

Система ФСПС состоит из трех элементов:

(1) *Функциональная подсистема* (такая, как система ФФП).

(2) *Состояние* D, представляющее собой множество определений функциональной подсистемы.

(3) Множество *правил перехода,* описывающих, как вход-

ные данные преобразуются в выходные и как изменяется состояние D.

Язык программирования системы ФСПС совпадает с языком ее функциональной подсистемы. (Всюду далее мы будем предполагать, что последняя является системой ФФП.) Итак, системы ФСПС могут использовать стиль программирования ФП, который мы уже обсуждали. Функциональная подсистема не может изменять состояние D, и оно не изменяется в ходе вычисления выражения. Новое состояние вычисляется одновременно с выходными данными и заменяет собой старое состояние в момент выдачи выходных данных. (Напоминаю, что множество определений D является последовательностью ячеек; имя ячейки — это имя определяемой функции, а ее содержимое — это определяющее выражение. Впрочем, некоторые ячейки могут именовать данные, а не функции; имя n данных будет употребляться в выражении $\uparrow n$ (доставка n), тогда как имя функции будет использоваться в качестве самого оператора.)

Ниже мы приводим правила переходов для элементарной системы ФСПС, которую мы используем для примеров программ. Пожалуй, это простейшие из многих возможных правил переходов, которые могли бы определять поведение широкого разнообразия систем ФСПС.

14.3.1. Правила переходов для элементарной системы ФСПС. Когда система принимает входные данные x, она формирует применение $(SYSTEM : x)$, а затем занимается получением его значения в подсистеме ФФП с использованием текущего состояния D в качестве множества определений. $SYSTEM$ — это особое имя некоторой функции, определенной в D (т. е. это «системная программа»). Обычно результатом является пара

$$\mu(SYSTEM : x) = \langle o, d \rangle$$

где o — это выходные данные системы, которые являются результатом обработки входных данных, а d становится новым состоянием D для следующих входных данных системы. Обычно состояние d будет копией или частично измененной копией старого состояния. Если $\mu(SYSTEM : x)$ не является парой, то на выходе получается сообщение об ошибке и состояние остается неизменным.

14.3.2. Правила перехода: исключительные условия и начальные действия. Коль скоро входные данные были получены, наша система не будет принимать других входных данных (кроме $\langle RESET, x \rangle$, см. ниже), пока не будут выданы входные данные и не будет установлено новое состояние, если оно возникнет. Система примет входные данные $\langle RESET, x \rangle$ в любой

момент. Существует два варианта: (а) если *SYSTEM* определена в текущем состоянии D, то система прерывает свои текущие вычисления, не изменяя D, и интерпретирует x как новые обычные входные данные; (б) если *SYSTEM* не определена в D, то x присоединяется к D в качестве первого элемента. (На этом завершается полное описание правил переходов для нашей элементарной системы ФСПС.)

Если *SYSTEM* определена в D, то она всегда может воспрепятствовать любому изменению в своем собственном определении. В противном случае обычные входные данные x породят $\mu(SYSTEM : x) = \perp$ и правила перехода приведут к сообщению об ошибке без изменения состояния. С другой стороны, входные данные $\langle RESET, \langle CELL, SYSTEM, s\rangle\rangle$ определят *SYSTEM* как s.

14.3.3. Программный доступ к состоянию; функция $\rho DEFS$. Требуется, чтобы наша подсистема ФФП обладала одной новой примитивной функцией определений, именуемой *DEFS* и такой, что для любого объекта $x \neq \perp$

$$\text{defs} : x = \rho DEFS : x = \text{D}$$

где D — текущее состояние и множество определения системы ФСПС. Эта функция представляет программам доступ ко всему состоянию для любой цели, в том числе для особо важной цели вычисления следующего состояния.

14.4. ПРИМЕР СИСТЕМНОЙ ПРОГРАММЫ

Данное выше описание нашей элементарной системы ФСПС, а также подсистемы ФФП, примитивов ФП и функциональных форм из предыдущих разделов специфицирует полную исторически чувствительную вычислительную систему. Ее поведение на входе и выходе ограничивается простыми правилами перехода, но в остальных отношениях это мощная система, коль скоро она оснащена подходящим множеством определений. В качестве примера ее использования мы опишем небольшую системную программу, ее реализацию и работу.

В нашем примере системной программы мы будем оперировать запросами и обновлениями для поддерживаемого ею файла, вычислять выражения ФФП, организовывать выполнение программ любого пользователя, не угрожающих сохранности файла или состояния, и разрешать некоторым избранным пользователям изменять множество описаний и саму системную программу. Все допустимые для нее входные данные будут иметь форму $\langle key, input\rangle$, где *key* (ключ) — это код, который определяет как класс входных данных (*изменение в системе, выражение, программа, запрос, обновление*), так и личность

пользователя и его допуск к использованию системы применительно к заданному классу входных данных. Мы не станем специфицировать формат для ключа. *Вход* представляет собой соответственно входные данные того класса, который задается *ключом.*

14.4.1. Общий план системной программы. Состояние D нашей системы ФСПС будет содержать описания всех непримитивных функций, нужных для системной программы и для программ пользователей. (Каждое описание находится в некой ячейке из последовательности D.) Кроме того, имеется ячейка из D с именем *FILE*, содержащая *файл*, который поддерживается системой. Мы дадим описания ФП для функций, а потом покажем, как передавать их в систему в форме ФФП. Правила перехода делают входные данные операндом для *SYSTEM*, но мы планируем использовать именные функции для обращения к данным, так что первым делом мы создадим для входных данных две ячейки с именами *KEY* и *INPUT*, содержащие *ключ* и *вход*, и присоединим их к D. Эта последовательность ячеек содержит по одной ячейке для ключа, входа и файла; она будет операндом нашей главной функции, называемой подсистемой. Подсистема может тогда получить *ключ*, применив к своему операнду операцию $\uparrow KEY$ и т. д. Итак, определение

Def система≡pair → подсистема ∘f; [*NONPAIR*, defs]

где

$f \equiv \downarrow INPUT \circ \lceil 2, \downarrow \mathrm{KEY} \circ [1, \mathrm{defs}]]$

побуждает систему выдать *NONPAIR* и сохранить состояние неизменным, если входные данные не являются парой. В противном случае, если это ⟨*key, input*⟩, то

$$f : \langle key,\ input\rangle = \langle\langle CELL,\ INPUT,\ input\rangle, \langle CELL,\ KEY,\ key\rangle, d_1, \ldots, d_n\rangle$$

где $D = \langle d_1, \ldots, d_n\rangle$. (Мы могли бы сконструировать иной операнд ровно с тремя ячейками для *ключа, входа и файла.* Мы не поступили таким образом, потому что реальные программы в отличие от подсистемы содержат много именных функций, относящихся к данным, которые содержатся в состоянии, и эта «стандартная» конструкция операнда пригодна для таких случаев.)

14.4.2. Функция «подсистема». Теперь мы дадим описание ФП для функции подсистемы, а затем кратко поясним шесть ее вариантов и вспомогательные функции.

Def подсистема ≡

является-изменением-системы ∘↑*KEY*→ [изменение-отчета, применить]∘[↑*INPUT*, defs];
является-выражением ∘↑*KEY*→[↑*INPUT*, defs];
является-программой ∘↑*KEY*→проверка-системы ∘ применить ∘(↑*INPUT*, defs];
является-запросом ∘↑*KEY*→[ответ-на-запрос ∘[↑*INPUT*, ↑*FILE*], defs];
является-обновлением ∘↑*KEY*→
сообщение-об-обновлении, ↓*FILE*∘ [обновить, defs]]∘ [↑*INPUT*, ↑*FILE*];
[сообщение-об-ошибке ∘[↑*KEY*, ↑*INPUT*], defs].

В этой подсистеме имеются пять фраз "*p*→*f*" и заключительная функция по умолчанию для всех шести классов входных данных; интерпретации всех классов приводятся ниже. Напоминаю, что операндом является последовательность ячеек, содержащая *key*, *input*, *file*, а также все определенные функции из D, и что подсистема: *operand*=⟨*output*, *newstate*⟩.

Неверные входные данные. В этом случае результат задается последней функцией из определения, когда ключ не удовлетворяет ни одной из предшествующих фраз. Выходными данными является сообщение об ошибке: ⟨*key*, *input*⟩. Состояние остается неизменным, так как оно задается посредством defs : *operand*=D. (Мы предоставляем читателю самому придумать, что именно функция сообщение-об-ошибке должна генерировать из его операнда.)

Входные данные типа изменение-системы. Когда имеет место изменение-системы ∘↑*KEY* : *operand*=изменение-системы : *key*==*T*,

то ключ *key* указывает, что пользователь имеет допуск для внесения изменения в систему, а также то, что *input*=↑*INPUT* : *operand* представляет функцию *f*, которая должна быть применена к D для получения нового состояния *f* : D. (Разумеется, *f* : *D* может оказаться бесполезным новым состоянием; на него не налагается никаких ограничений.) Выходом является сообщение, а именно сообщение-об-обновлении ⟨*input*, D⟩.

Входные данные типа выражение. Когда является-выражением : *key*=*T*, то система понимает, что результатом должно быть значение ФФП-выражения *input*; ↑*INPUT* : *operand* порождает этот результат, и проводится вычисление, как для всяких выражений. Состояние остается неизменным.

Входные данные типа программы и самозащита системы. Когда является-программой : $key=T$, то и выходные данные, и новое состояние задаются так: $(\rho\ input)$: D = ⟨*output*, *newstate*⟩. Если *newstate* (новое состояние) содержит файл в надлежащем условии, а также определения системы и другие защищенные функции, то проверка-системы: ⟨*output*, *newstate*⟩ = ⟨*output*, *newstate*⟩. В противном случае проверка-системы: ⟨*output*, *newstate*⟩ = ⟨*error-report*, D⟩.

Хотя входные данные *program* могут вносить значительные, возможно губительные, изменения в состояние, когда порождается *newstate*, все же в действии проверка-системы можно использовать любой критерий, чтобы позволить ему стать действительно новым состоянием либо сохранить прежнее. Более изощренная проверка-системы могла бы корректировать только запрещенные изменения состояния. Такого рода функции осуществимы, потому что всегда возможен доступ к прежнему состоянию для сравнения с намечающимся новым состоянием и контроль окончательного разрешения замены состояния.

Входные данные типа запросов к файлам. Если является-запросом : $key=T$, то разрабатывается функция ответ-на-запрос для получения выходных данных, являющихся ответом на входные данные запроса от его операнда ⟨*input*, *file*⟩.

Входные данные типа обновления файлов. Если является-обновлением : $key=T$, то вход специфицирует изменение состояния, понимаемое функцией *update* (обновление), которая вычисляет *updated-file* = обновление : ⟨*input*, *file*⟩. Таким образом, для $\downarrow FILE$ операндом является ⟨*updated-file*, *D*⟩, и поэтому $\downarrow FILE$ запоминает обновленный файл в ячейке *FILE* в новом состоянии. Остальная часть состояния не изменяется. Функция сообщение-об-обновлении генерирует выходные данные на основании своего операнда ⟨*input*, *file*⟩.

14.4.3. Реализация системной программы. Мы описали функцию, называемую системой, посредством некоторых определений ФП (пользуясь вспомогательными функциями, поведение которых подробно не описывалось, а только указывалось). Предположим, что имеются определения ФП для всех требующихся непримитивных функций. Тогда можно преобразовать каждое определение, чтобы получить соответствующее имя и содержимое ячейки в D (разумеется, само это преобразование выполнялось бы другой, лучшей системой). Преобразование завершается заменой всякого имени функции ФП на эквивалентный атом (например, обновление приобретает вид *UPDATE*) и заменой функциональных форм на последовательности, в которых первый член является управляющей функцией для конк-

ретной формы. Так, $\downarrow FILE \circ [\text{update, defs}]$ преобразуется в

$\langle COMP, \langle STORE, FILE\rangle, \langle CONS, UPDATE, DEFS\rangle\rangle$

и функция ФП является такой же, как функция, представляемая объектом ФФП, при условии, что обновление $\equiv \rho UPDATE$ и *COMP*, *STORE* и *CONS* представляют управляющие функции для композиции, запоминания и конструкции.

Все определения ФП, нужные для нашей системы, могут быть преобразованы в ячейки, как отмечалось выше; в результате получается последовательность D_0. Мы предполагаем, что у системы ФСПС имеется пустое состояние, с которого можно начать; поэтому *SYSTEM* не определяется. Мы хотим дать начальное определение *SYSTEM*, чтобы эта система реализовывала свои следующие данные как состояние; после этого мы можем ввести D_0 и будут реализованы все наши описания, включая саму нашу программу-систему. Для осуществления этого введем первые входные данные:

$\langle RESET, \langle CELL, SYSTEM, loader\rangle\rangle$

где

$loader \equiv \langle CONS, \langle\langle CONST, DONE\rangle, ID\rangle.$

Тогда, согласно правилу перехода для *RESET*, когда *SYSTEM* не определена в D, ячейка в наших входных данных помещается в начало $D = \varnothing$, определяя тем самым $\rho SYSTEM \equiv \rho loader \equiv$ $\equiv [\overline{DONE}, \text{id}]$. Наши вторые входные данные — это D_0, т. е. то множество определений, которое мы хотим сделать состоянием. Обычно правило перехода побуждает систему ФСПС вычислить

$\mu(SYSTEM : D_0)[\overline{DONE}, \text{id}] : D_0 = \langle DONE, D_0\rangle.$

Таким образом, выходным значением для ваших вторых входных данных является *DONE*, новое состояние есть D_0, а $\rho SYSTEM$ теперь представляет собой нашу системную программу (которая принимает входные данные только вида $\langle key, input\rangle$).

Наша следующая задача состоит в том, чтобы загрузить файл (мы получаем начальное значение *file*). Чтобы загрузить его, мы вводим *program* во вновь реализованную систему, которая содержит *file* как константу и запоминает эту константу в состоянии; входные данные имеют вид

$\langle program\text{-}key, [\overline{DONE}, store\text{-}file]$

где

$\rho store\text{-}file \equiv \downarrow FILE \circ [\overline{file}, \text{id}].$

Program-key идентифицирует [*DONE*, *store-file*] как программу, которую нужно применить к состоянию D_0 для получения

выходных данных и нового значения D_1, которое имеет вид

$$pstore\text{-}file : D_0 = \downarrow FILE \circ [\overline{file}, \mathrm{id}] : D_0$$

или D_0 с ячейкой, содержащей в своем начале *file*. Выходные данные — это $\overline{DONE} : D_0 = DONE$. Мы предполагаем, что проверка-системы оставит пару $\langle DONE, D_1 \rangle$ неизменной. Выше применялись выражения ФП вместо обозначаемых ими объектов ФФП, например $\overline{DONE}$ вместо $\langle CONST, DONE \rangle$.

14.4.4. Применение системы. Мы ничего не сказали о том, как структурируются системы, запросы или обновления, и поэтому не можем привести подробный пример операций с файлом. Однако из структуры подсистемы ясно видно, как ответ системы на запросы и обновления зависит от функций ответ-на-запрос, обновление и сообщение-об-обновлении.

Предположим, что матрицы *m*, *n* с именами *M* и *N* запомнены в D и что описанная выше функция MM определена в D. Тогда входные данные

$$\langle expression\text{-}key, (MM \circ [\uparrow M, \downarrow N] \circ DEFS : \#) \rangle$$

породят в качестве результата произведение этих двух матриц и неизменное состояние. *Expression-key* идентифицирует применение как выражение, которое нужно вычислить, и поскольку $defs : \# = D$ и $[\uparrow M, \uparrow N] : D = \langle m, n \rangle$, то значением выражения является результат $MM : \langle m, n \rangle$, представляющий собой выходные данные.

Наша миниатюрная системная программа не содержит средств передачи управления программе пользователя для обработки многих потоков входных данных, но не представило бы большого труда снабдить ее такими средствами с сохранением мониторинга над программой пользователя с возможностью обратной передачи управления.

14.5. ВАРИАНТЫ СИСТЕМ ФСПС

Существенным обобщением предложенных выше систем ФСПС была бы система, обеспечивающая комбинационные формы, т. е. «системные формы» для построения новых систем ФСПС из более простых подсистем ФСПС. Иначе говоря, системная форма использовала бы системы ФСПС как параметры и генерировала бы новую систему ФСПС точно так же, как функциональная форма берет в качестве параметров функции и генерирует новые функции. Эти системные формы были бы сходны по своим свойствам с функциональными формами и стали бы операциями удобной «алгебры систем» во многих отношениях аналогично тому, как функциональные формы представ-

ляют собой «операции» алгебры программ. Однако проблема отыскания удобных системных форм гораздо более трудна, поскольку они должны обрабатывать *RESETS*, сопоставлять входные данные с выходными и комбинировать исторически чувствительные системы, а не фиксированные функции.

Кроме того, полезность или нужность системных форм менее очевидна, чем в случае функциональных форм, когда она имеет существенное значение для построения большого разнообразия функций из исходного множества примитивов, причем даже при отсутствии системных форм средства построения систем ФСПС уже настолько богаты, что можно было бы строить фактически любую систему (при общих свойствах ввода и вывода, предоставляемых заданной схемой ФСПС). Быть может, системные формы оказались бы полезными для построения систем со сложными упорядочениями входных и выходных данных.

14.6. ЗАМЕЧАНИЯ О СИСТЕМАХ ФСПС

Как я пытался показать выше, возможны бесчисленные вариации составных частей системы ФСПС — в том, как она работает, как обрабатывает ввод и вывод, как и когда порождает новые состояния и т. д. В любом случае ряд замечаний уместен для любой осмысленной системы ФСПС:

(а) Изменение состояния происходит однократно для каждого большого вычисления и может обладать полезными математическими свойствами. Изменения состояний не связаны с мелкими подробностями вычисления, как в традиционных языках, так что устраняется лингвистическая «узость фон Неймана». Для связи с состоянием не требуется сложного «кабеля» или протоколов.

(б) Программы пишутся на аппликативном языке, и он может включать большое разнообразие изменяемых частей, мощь и гибкость которых превосходит возможности любого из существующих языков фон Неймана. Стиль «слово за словом» заменяется на функциональный стиль, отсутствует разделение программирования на мир выражений и мир операторов. Программы могут анализироваться и оптимизироваться посредством алгебры программ.

(в) Поскольку состояние не может измениться в ходе вычисления выражения система: x, то не возникает побочных эффектов. Поэтому независимые применения могут вычисляться параллельно.

(г) Я полагаю, что посредством описания подходящих функций можно вводить в любой момент важные новые свойства, используя ту же структуру. Такие свойства должны реализовы-

ваться в рамках структуры языка фон Неймана. Я имею в виду такие возможности, как «регистры памяти» с большим разнообразием систем именования, типы и проверки типов, взаимодействующие параллельные процессы, недетерминированность и конструкции «охраняемых команд» Дейкстры [8], а также улучшенные методы структурного программирования.

(д) Структура системы ФСПС заключает в себе синтаксис и семантику лежащей в основе функциональной системы в сочетании с намеченной выше системной структурой. По современным стандартам это является очень слабой структурой для языка, и никаких других фиксированных частей система не содержит.

14.7. СИСТЕМЫ ИМЕНОВАНИЯ В МОДЕЛЯХ ФСПС И ФОН НЕЙМАНА

Как отмечалось в разд. 13.3.3, в системе ФСПС именование осуществляется функциями. Можно описать много полезных функций для доступа к памяти и изменения ее содержимого (например, занести на стек, снять со стека, чистить, выбрать и т. д.). Все эти определения и соответствующие им системы именования могут быть введены без изменения структуры ФСПС. В одной и той же программе могут использоваться различные виды «регистров памяти» (например, с ячейками, снабженными типом) со своими системами именования. Ячейка одного регистра может содержать другой регистр целиком.

Важная особенность систем именования ФСПС состоит в том, что они реализуют функциональную природу имен (система GEDANKEN Рейнолдса [19] до некоторой степени обеспечивает то же самое в рамках структуры фон Неймана). Функции-имена могут объединяться и комбинироваться с другими функциями посредством функциональных форм. В языках фон Неймана, наоборот, функции и имена обычно представляют собой не связанные между собой понятия, и сходство природы функций и имен почти полностью игнорируется и не используется, потому что (а) имена не могут применяться как функции; (б) отсутствуют общие средства комбинирования имен с другими именами и функциями; (в) объекты, к которым применяются имена-функции (регистры памяти), недоступны в качестве обычных объектов.

Возможно, одна из основных слабостей языков фон Неймана состоит в их неспособности интерпретировать имена как функции. Во всяком случае, возможность употреблять имена как функции и регистры памяти как объекты может оказаться полезной и важной концепцией программирования, которая заслуживает тщательной разработки.

15. ЗАМЕЧАНИЯ О ПРОЕКТИРОВАНИИ КОМПЬЮТЕРОВ

Из-за преобладания языков фон Неймана проектировщики имели в своем распоряжении немного интеллектуальных моделей для практической разработки компьютеров вне рамок вариантов компьютера фон Неймана. Другим классом исторически чувствительных моделей являются модели потоков данных [1, 7, 13]. Правила подстановок языков, основанных на лямбда-исчислении, ставят перед проектировщиком вычислительных машин серьезные проблемы. Берклинг [3] разработал модифицированное лямбда-исчисление с тремя видами применений, в котором нет необходимости переименовывать переменные. Он спроектировал машину для вычисления выражений этого языка. Требуется дополнительная практика, чтобы показать, насколько надежной основой является этот язык для эффективного стиля программирования и сколь эффективной может стать машина Берклинга.

Маго [15] разработал новую функциональную машину, которая строится из идентичных компонентов (двух видов). Она непосредственно вычисляет снизу вверх функциональные выражения типа ФП и других типов. В ней отсутствует «память» фон Неймана и нет регистра адреса, а следовательно, нет и узости; она способна параллельно вычислять много приложений; ее встроенные операции напоминают операторы ФП в большей степени, чем операции компьютера фон Неймана. Из всех известных мне конструкций эта наиболее отдалилась от компьютера фон Неймана.

Имеются многочисленные свидетельства того, что функциональный стиль программирования может по своей мощи превзойти стиль фон Неймана. Поэтому для программистов имеет большое значение разработка нового класса исторически чувствительных моделей вычислительных систем, которые воплощают такой стиль и не страдают неэффективностью, по-видимому, присущей системам, основанным на лямбда-исчислении. Только когда такие системы и их функциональные языки докажут свое превосходство над традиционными языками, мы получим экономическую основу для разработки нового вида компьютера, который сможет наилучшим образом реализовать их. Только тогда мы, вероятно, сможем в полной мере использовать большие интегральные схемы в архитектуре компьютеров, не ограниченных узостью фон Неймана.

16. ИТОГИ

Пятнадцать предыдущих разделов этой статьи могут быть подытожены следующим образом:

Раздел 1. Традиционные языки программирования громоздки, сложны и негибки. Их ограниченные выразительные возможности не оправдывают их пространности и дороговизны.

Раздел 2. Модели вычислительных систем, лежащие в основе языков программирования, грубо говоря, распадаются на три класса: (а) простые операционные модели (например, машины Тьюринга), (б) функциональные модели (например, лямбда-исчисление) и (в) модели фон Неймана (например, традиционные компьютеры и языки программирования). Каждому классу моделей присуще свое существенное затруднение: программы класса (а) не структурируемы; модели класса (б) не в состоянии передавать информацию от одной программы к следующей; основания моделей класса (в) не обладают практически полезными свойствами, а программы концептуально бесплодны.

Раздел 3. Компьютеры фон Неймана строятся вокруг узкого места — трубки, передающей слово за словом и соединяющей центральное процессорное устройство с памятью. Поскольку программа должна выполнять все свои изменения в памяти посредством перекачивания множества слов туда и обратно через эту «узость фон Неймана», мы воспитаны на стиле программирования, в котором основное внимание уделяется организации потока слов через эту узость, а не более крупным концептуальным единицам наших проблем.

Раздел 4. Традиционные языки основываются на стиле программирования для компьютера фон Неймана. Так, переменные=ячейки памяти; операторы присваивания=выборка, запоминание и арифметика; операторы управления=команды передачи управления и проверки условия. Символ «: =» является лингвистической «узостью фон Неймана». Программирование на традиционном (фон Неймана) языке по-прежнему озабочено потоком слов (по одному за раз) через слегка усовершенствованную узость. К тому же языки фон Неймана расщепляют программирование на мир выражений и мир операторов. Первый из них упорядочен, а второй хаотичен и лишь слегка упрощен структурным программированием, которое оставило незатронутыми основные проблемы самого расщепления и пословного стиля традиционных языков.

Раздел 5. В этом разделе сравниваются программа фон Неймана и функциональная программа для вычисления внутреннего произведения. Иллюстрируется ряд трудностей, присущих первой программе, и ряд преимуществ второй: например, программа фон Неймана является итеративной и пословной, работает только с двумя векторами a и b заданной длины n и мо-

жет стать общей только за счет использования описания процедуры, которое характеризуется сложной семантикой. Функциональная программа является неитеративной, работает с векторами как с объектами, более иерархически сконструирована; она вполне общая и создает операции «внутреннего хозяйства» композицией операторов внутреннего хозяйства более высокого уровня. Она не именует свои аргументы и поэтому не требует описания процедуры.

Раздел 6. Язык программирования включает в себя структуру вместе с некоторыми изменяемыми частями. Структура языка фон Неймана требует, чтобы большинство средств были встроены в нее; она может освоить только ограниченные изменяемые части (например, определяемые пользователем процедуры), потому что для всех нужд изменяемых частей, а также для всех возможностей, встроенных в структуру, должно быть предусмотрено детальное обеспечение в «состоянии» и в его правилах перехода. Структура фон Неймана является столь негибкой именно потому, что ее семантика слишком сильно привязана к состоянию: всякая подробность вычислений изменяет состояние.

Раздел 7. Выразительная сила изменяемых частей в языках фон Неймана мала; именно по этой причине большинство средств языка должны быть встроены в структуру. Недостаточность выразительной силы следует из того, что в языках фон Неймана отсутствует возможность эффективно использовать для построения программ комбинационные формы, а это, в свою очередь, следует из расщепления языка на мир выражений и мир операторов. Комбинационные формы лучше всего проявляют себя в выражениях, но в языках фон Неймана выражение способно порождать только одиночное слово; поэтому выразительная сила в мире выражений по большей части теряется. Другим препятствием применению комбинационных форм являются хитроумные соглашения об именовании.

Раздел 8. APL — это первый не основанный на лямбда-исчислении язык, который работает не «слово за словом» и использует функциональные комбинационные формы. Однако в нем все еще сохраняются многие из проблем, присущих языкам фон Неймана.

Раздел 9. Язык фон Неймана не представляет удобных средств для рассуждений о программах. Аксиоматические и денотационные семантики являются точным инструментарием для описания и понимания традиционных программ, но они лишь позволяют рассуждать о программах и не делают их более удобными. В отличие от языков фон Неймана язык обычной

алгебры удобен как для формулирования ее законов, так и для преобразования уравнений в их решения, и все это в рамках одного «языка».

Раздел 10. В исторически чувствительном языке одна программа может влиять на поведение другой, последующей, за счет изменения состояния части памяти, которая сохраняется системой. Любой такой язык требует некоторой семантики переходов состояний. Но ему не нужна семантика, тесно привязанная к состояниям, в которой состояние изменяется с каждой подробностью вычислений. Функциональные системы переходов состояний (ФСПС) предлагаются в качестве исторически чувствительной альтернативы для систем фон Неймана. Они характеризуются: (а) свободно привязанной семантикой переходов состояний, в которой переход происходит лишь однажды для большого вычисления; (б) простыми состояниями и правилами перехода; (в) базовой функциональной системой с простой семантикой «сведения», (г) языком программирования и правилами перехода состояний, основывающимися на базовой функциональной системе и ее семантике. В следующих четырех разделах излагаются элементы этого подхода к проектированию не-фон-неймановских языков и систем.

Раздел 11. Описывается класс систем неформального функционального программирования (ФП) без использования переменных. Каждая система строится из объектов, функций, функциональных форм и определений. Функции отображают объекты на объекты. Функциональные формы комбинируют существующие функции для формирования новых. В этом разделе перечисляются примеры примитивных функций и функциональных форм и приводятся примеры программ. Обсуждаются ограничения и преимущества систем ФП.

Раздел 12. Описывается «алгебра программ», в которой переменными являются функции систем ФП, а операции — это функциональные формы системы. Список двадцати четырех законов алгебры сопровождается примером доказательства эквивалентности неитеративной программы умножения матриц и рекурсивной программы для той же цели. В следующем подразделе формулируются результаты двух «теорем о разложении», которые «решают» два класса уравнений. Эти решения выражают «неизвестную» функцию из таких уравнений в виде бесконечного условного выражения, которое устанавливает ситуационное описание ее поведения и непосредственно дает необходимые и достаточные условия для завершения. Эти результаты используются для вывода «теоремы рекурсии» и «теоремы итерации», которые обеспечивают готовые к употреблению

выражения для некоторых довольно общих и полезных классов «линейных» уравнений. Примеры применения этих теорем включают: (а) доказательства корректности для рекурсивной и итеративной факториальных функций и (б) доказательство эквивалентности двух итеративных программ. Заключительный пример относится к «квадратному» уравнению и содержит доказательство того, что его решение представляет собой идемпотентную функцию. В следующем подразделе приводятся доказательства двух теорем о разложении.

Ассоциированная с системами ФП алгебра сопоставляется с соответствующими алгебрами для лямбда-исчисления и других функциональных систем. Сопоставление показывает некоторые преимущества несколько ограниченных систем ФП по сравнению с гораздо более мощными классическими системами. Ставятся некоторые вопросы относительно алгоритмического сведения функций бесконечных выражений и относительно использования этой алгебры в различных схемах «ленивого вычисления».

Раздел 13. В этом разделе описываются системы формального функционального программирования, которые обобщают и уточняют поведение систем ФП. Их семантики проще семантик классических систем, и их непротиворечивость можно показать простым рассуждением.

Раздел 14. В этом разделе сравниваются структуры Алгола и функциональных систем переходов состояний (ФСПС). Описывается система ФСПС, использующая систему ФФП в качестве своей функциональной подсистемы. Для этой системы вводятся простое состояние и правила перехода. Описывается малая самозащищающаяся системная программа для данной системы ФСПС и рассматривается возможность ее реализации и применения для поддержки и организации вычислений по программам пользователей. Кратко обсуждаются варианты систем ФСПС и функциональных систем, которые могут быть описаны и использованы в рамках системы ФСПС.

Раздел 15. В этом разделе кратко рассматриваются работа по проектированию функциональных компьютеров и потребность в разработке и тестировании более практичных моделей функциональных систем как будущей основы такого проектирования.

БЛАГОДАРНОСТИ

В прежней работе, связанной с этой статьей, я получил весьма значительную помощь и много советов от Пола Макджонса и Барри К. Розена. Я также получил очень ценную по-

мощь и советы при подготовке этой статьи. Дж. Н. Грей исключительно щедро делился своими знаниями и потратил много времени на рецензирование первого варианта статьи. Стефен Н. Зиллис тоже тщательно прочитал его. Оба предложили много важных советов и замечаний на этом трудном этапе. Я пользуюсь приятной возможностью выразить им свою признательность. Для меня были полезны также обсуждения первого варианта с Ренальдом Фейджином, Полом Р. Мак-Джонсом и Джеймсом Моррисом. Фейджин предложил ряд усовершенствований доказательства теорем.

Поскольку большая часть статьи содержит технический материал, я попросил двух разных специалистов по информатике прорецензировать третий вариант. Дэвид Грис и Джон Рейнольдс оказались настолько любезны, что взяли на себя этот обременительный труд. Оба предоставили мне большие подробные списки исправлений и общих замечаний, которые привели ко многим большим и малым улучшениям этого последнего варианта (прорецензировать который им не представилось возможности). Я искренне благодарен за затраченные ими время и усилия.

Наконец, я послал копии третьего варианта Джуле Маго, Питеру Науру и Джону Уильямсу. Они любезно откликнулись, прислав ряд исключительно ценных замечаний и исправлений. Джеффри Франк и Дэйв Тоули из Университета Северной Каролины ознакомились с копией, посланной мною Маго, и указали существенную ошибку в описании семантических функций систем ФФП. Всем им я глубоко признателен за оказанную мне помощь.

ЛИТЕРАТУРА

1. Arvind and Gostelow K. P. A new interpreter for data flow schemas and its implications for computer architecture. Tech. Rep. No. 72, Dept. Comptr. Sci., U. of California, Irvine, Oct. 1975.
2. Backus J. Programming language semantics and closed applicative languages. Conf. Record ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, 71—86.
3. Berkling K. J. Reduction languages for reduction machines. Interner Bericht ISF—76—8, Gesellschaft fur Mathematik und Datenverarbeitung MBH, Sept. 1976.
4. Burge W. H. Recursive Programming Techniques. Addison-Wesley, Reading Mass., 1975.
5. Church A. The Calculi of Lambda-Conversion. Princeton U. Press, Princeton, N. J., 1941.
6. Curry H. B. and Feys R. Combinatory Logic, Vol. I. North-Holland Pub. Co., Amsterdam, 1958.
7. Dennis J. B. First version of a data flow procedure language. Tech. Mem. No. 61, Lab. for Comptr. Sci., M. I. T., Cambridge, Mass., May 1973.
8. Dijkstra E. W. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N. J., 1976. [Имеется перевод: Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978. — 274 с.

9. Friedman D. P. and Wise D. S. CONS should not evaluate its arguments. In Autumata, Languages and Programming, S. Michaelson and R. Milner, Eds., Edinburgh U. Press, Edinburgh, 1976, pp. 257—284.
10. Henderson P. and Morris J. H. Jr. A lazy evaluator. Conf. Record 3rd ACM Symp. on Principles of Programming Languages, Atlanta, Ga., Jan. 1976, pp. 95—103.
11. Hoare C. A. R. An axiomatic basis for computer programming. Comm. ACM 12, 10 (Oct. 1969), 576—583.
12. Iverson K. A. Programming Language. Wiley, New York, 1962.
13. Kosinski P. A data flow programming language. Rep. RS 4264, IBM T. J. Watson Research Ctr., Yorktown Heigths, N. Y., March 1973.
14. Landin P. J. The mechanical evaluation of expressions. Computer J. 6, 4 (1964), 308—320.
15. Mago G. A. A network of microprocessors to execute reduction languages. To appear in Int. J. Comptr. and Inform. Sci.
16. Manna Z., Ness S., and Vuillemin J. Inductive methods for proving properties of programs. Comm. ACM 16, 8 (Aug. 1973), 491—502.
17. McCarthy J. Recursive functions of symbolic expressions and their computation by machine, Pt. 1. Comm. ACM 3, 4 (April 1960), 184—195.
18. McJones P. A. Church—Rosser property of closed applicative languages. Rep. RJ 1589, IBM Res. Lab., San Jose, Calif., May 1975.
19. Reynolds J. C. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. Comm. ACM 13, 5 (May 1970), 308—318.
20. Reynolds J. C. Notes on a lattice-theoretic approach to the theory of computation. Dept. Syst. and Inform. Sci., Syracuse U., Syracuse, N. Y., 1972.
21. Scott D. Outline of a mathematical theory of computation. Proc. 4th Princeton Conf. on Inform. Sci. and Syst., 1970.
22. Scott D. Lattice-theoretic models for various types-free calculi. Proc. 4th Int. Congress for Logic, Methodology, and the Philosophy of Science, Bucharest, 1972.
23. Scott D. and Strachey C. Towards a mathematical semantics for computer languages. Proc. Symp. on Comptrs. and Automata, Polytechnic Inst. of Brooklyn, 1971.