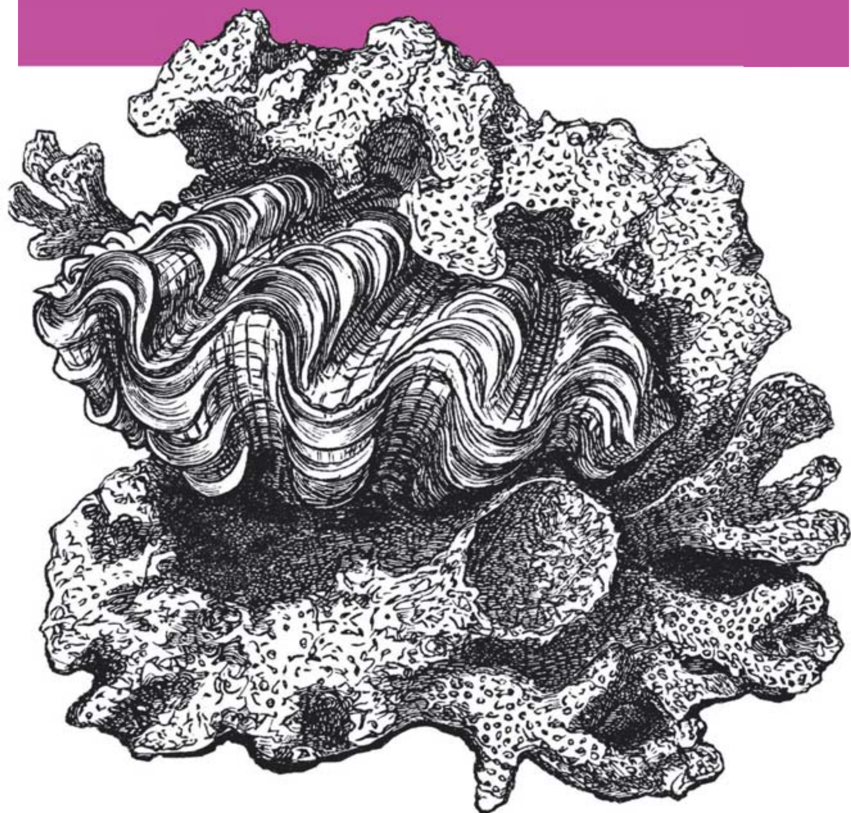
Теория вычислений для программистов



Том Стюарт

ДМК пресс издательство

O'REILLY®

УДК 004.421.2
ББК 32.973-018
С759

Стюарт Т.

С759 Теория вычислений для программистов / Пер. с анг. А. А. Слинкин. – М.: ДМК Пресс, 2014. – 384 с.: ил.

ISBN 978-5-94074-979-0

Наконец-то появился увлекательный и практичный способ изучать теорию вычислений и проектирование языков программирования!

В этой книге теоретическая информатика излагается в хорошо знакомом вам контексте, что поможет оценить, почему ее идеи важны и как они отражаются на том, чем программист изо дня в день занимается на работе.

Вместо математической нотации или незнакомого академичного языка программирования типа Haskell или Lisp в этой книге для объяснения формальной семантики, теории автоматов и функционального программирования вкупе с лямбда-исчислением применяется язык Ruby, сведенный к минимуму.

Издание предназначено для программистов любой квалификации, знакомых хотя бы с одним из современных языков, но не имеющих формальной подготовки в информатике.

УДК 004.421.2
ББК 32.973-018

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-449-32927-3 (анг.)
ISBN 978-5-94074-979-0 (рус.)

Copyrigth © 2013 Tom Stuart
© Оформление, перевод, ДМК Пресс, 2014

Содержание

Предисловие 10
Для кого предназначена эта книга 10
Графические выделения 10
О примерах кода 11
Как с нами связаться 11
Благодарности 12

Глава 1. Все, что нужно знать о Ruby 14
Интерактивная оболочка Ruby 14
Значения 15
  Простые данные 15
Структуры данных 16
Процедуры 17
Поток управления 18
Объекты и методы 18
Классы и модули 20
Прочее 21
  Локальные переменные и присваивание 22
  Строковая интерполяция 22
  Инспектирование объектов 22
  Печать строк 23
  Методы с переменным числом аргументов 23
  Блоки 24
  Модуль Enumerable 25
  Класс Struct 26
  Партизанское латание 27
  Определение констант 28
  Удаление констант 28

Часть I. ПРОГРАММЫ И МАШИНЫ ... 30

Глава 2. Семантика программ ... 32

В чем смысл слова «смысл»? ... 33
Синтаксис ... 35
Операционная семантика ... 36
Семантика мелких шагов ... 37
    Выражения ... 39
    Предложения ... 50
    Корректность ... 60
    Приложения ... 61
Семантика крупных шагов ... 62
    Выражения ... 63
    Предложения ... 65
    Приложения ... 68
Денотационная семантика ... 70
    Выражения ... 71
    Предложения ... 75
    Сравнение способов определения семантики ... 76
    Приложения ... 77
Формальная семантика на практике ... 79
    Формализм ... 79
Поиск смысла ... 80
    Альтернативы ... 81
Реализация синтаксических анализаторов ... 82

Глава 3. Простейшие компьютеры ... 88

Детерминированные конечные автоматы ... 88
    Состояния, правила и входной поток ... 89
    Вывод ... 90
    Детерминированность ... 91
    Моделирование ... 92
Недетерминированные конечные автоматы ... 96
    Недетерминированность ... 96
    Свободные переходы ... 104
Регулярные выражения ... 108
    Синтаксис ... 109
    Семантика ... 112
    Синтаксический анализ ... 122
Эквивалентность ... 124
    Минимизация ДКА ... 134

Глава 4. Кое-что помощнее 136
Детерминированные автоматы с магазинной памятью 140
Память 140
Правила 142
Детерминированность 144
Моделирование 145
Недетерминированные автоматы с магазинной памятью 152
Моделирование 156
Неэквивалентность 159
Разбор с помощью автоматов с магазинной памятью 160
Лексический анализ 161
Синтаксический анализ 163
Применение на практике 168
Насколько мощнее? 169

Глава 5. Окончательная машина 172
Детерминированные машины Тьюринга 172
Память 173
Правила 176
Детерминированность 180
Моделирование 180
Недетерминированные машины Тьюринга 187
Максимальная мощность 188
Внутренняя память 189
Подпрограммы 192
Несколько лент 194
Многомерная лента 195
Машины общего назначения 196
Кодирование 198
Моделирование 200

Часть II. ВЫЧИСЛЕНИЯ И ВЫЧИСЛИМОСТЬ 201

Глава 6. Программирование на пустом месте 203
Имитация лямбда-исчисления 204
Работа с процедурами 205
Задача 207
Числа 209
Булевы значения 213

Предикаты 217
Пары 218
Операции над числами 219
Списки 228
Строки 231
Решение 234
Более сложные приемы программирования 238
Реализация лямбда-исчисления 245
Синтаксис 245
Семантика 247
Синтаксический разбор 253

Глава 7. Универсальность повсюду 256

Лямбда-исчисление 257
Частично рекурсивные функции 260
SKI-исчисление 266
Iota 276
Таг-системы 280
Циклические таг-системы 289
Игра «Жизнь» Конвея 300
Правило 110 303
Вольфрамова 2,3 машина Тьюринга 307

Глава 8. Невозможные программы 308

Факты как они есть 309
Универсальные системы могут выполнять алгоритмы 309
Программы могут замещать машины Тьюринга 313
Код – это данные 314
Универсальные системы могут зацикливаться 316
Программы могут ссылаться сами на себя 323
Разрешимость 329
Проблема остановки 331
Построение анализатора остановки 331
Это никогда работать не будет 334
Другие неразрешимые проблемы 339
Печальные следствия 342
Почему так происходит? 345
Жизнь в условиях невычислимости 346

Глава 9. Программирование в игрушечной стране 349
Абстрактная интерпретация 350
Планирование маршрута 351
Абстракция: умножение знаков 352
Аппроксимация и безопасность: сложение знаков 356
Статическая семантика 361
Реализация 363
Достоинства и ограничения 371
Приложения 374

Послесловие 376

Предметный указатель 378

Семантика мелких шагов

Итак, как спроектировать абстрактную машину и использовать ее для специфицирования операционной семантики языка программирования? Один из возможных способов – представить себе машину, которая вычисляет программу, действуя прямо по синтаксическим правилам и мелкими шагами *сворачивая* программу, так что на каждом шаге она становится ближе к конечному результату, что бы это ни значило.

Эти мелкие шаги свертки похожи на способ вычисления алгебраических выражений, который мы проходили в школе. Например, чтобы вычислить выражение `(1 × 2) + (3 × 4)`, мы должны:

1. Выполнить умножение в скобках слева (`1 × 2` равно `2`) и свернуть выражение в `2 + (3 × 4)`.
2. Выполнить умножение в скобках справа (`3 × 4` равно `12`) и свернуть выражение в `2 + 12`.
3. Выполнить сложение (`2 + 12` равно `14`), получив окончательный результат – `14`.

Мы можем назвать число 14 результатом, потому что дальше оно не сворачивается, – мы понимаем, что 14 – особый вид алгебраического выражения, *значение*, у которого есть самостоятельная семантика, так что с нашей стороны никакой дополнительной работы не требуется.

Эту неформальную процедуру можно превратить в операционную семантику, выписав формальные правила выполнения каждого шага свертки. Эти правила сами должны быть написаны на некотором языке (*метаязыке*), который обычно представляет собой математическую нотацию.

В этой главе мы исследуем семантику игрушечного языка программирования, назовем его SIMPLE[1].

Математическое описание семантики мелких шагов для языка SIMPLE выглядит следующим образом:

---

[1] Если хотите, можете считать это аббревиатурой «simple imperative language» (простой императивный язык).

$$\frac{\langle e_1, \sigma\rangle \leadsto_e e_1'}{\langle e_1 + e_2, \sigma\rangle \leadsto_e e_1' + e_2} \qquad \frac{\langle e_2, \sigma\rangle \leadsto_e e_2'}{\langle v_1 + e_2, \sigma\rangle \leadsto_e v_1 + e_2'}$$

$$\frac{}{\langle n_1 + n_2, \sigma\rangle \leadsto_e n} \text{ if } n = n_1 + n_2$$

$$\frac{\langle e_1, \sigma\rangle \leadsto_e e_1'}{\langle e_1 * e_2, \sigma\rangle \leadsto_e e_1' * e_2} \qquad \frac{\langle e_2, \sigma\rangle \leadsto_e e_2'}{\langle v_1 * e_2, \sigma\rangle \leadsto_e v_1 * e_2'}$$

$$\frac{}{\langle n_1 * n_2, \sigma\rangle \leadsto_e n} \text{ if } n = n_1 \times n_2$$

$$\frac{\langle e_1, \sigma\rangle \leadsto_e e_1'}{\langle e_1 < e_2, \sigma\rangle \leadsto_e e_1' < e_2} \qquad \frac{\langle e_2, \sigma\rangle \leadsto_e e_2'}{\langle v_1 < e_2, \sigma\rangle \leadsto_e v_1 < e_2'}$$

$$\frac{}{\langle n_1 < n_2, \sigma\rangle \leadsto_e \texttt{true}} \text{ if } n_1 < n_2 \qquad \frac{}{\langle n_1 < n_2, \sigma\rangle \leadsto_e \texttt{false}} \text{ if } n_1 \geq n_2$$

$$\frac{}{\langle x, \sigma\rangle \leadsto_e \sigma(x)} \text{ if } x \in \mathrm{dom}(\sigma)$$

$$\frac{\langle e, \sigma\rangle \leadsto_e e'}{\langle x = e, \sigma\rangle \leadsto_s \langle x = e', \sigma\rangle} \qquad \frac{}{\langle x = v, \sigma\rangle \leadsto_s \langle \texttt{do-nothing}, \sigma[x \mapsto v]\rangle}$$

$$\frac{\langle e, \sigma\rangle \leadsto_e e'}{\langle \texttt{if (}e\texttt{) \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}, \sigma\rangle \leadsto_s \langle \texttt{if (}e'\texttt{) \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}, \sigma\rangle}$$

$$\frac{}{\langle \texttt{if (true) \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}, \sigma\rangle \leadsto_s \langle s_1, \sigma\rangle} \qquad \frac{}{\langle \texttt{if (false) \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}}, \sigma\rangle \leadsto_s \langle s_2, \sigma\rangle}$$

$$\frac{\langle s_1, \sigma\rangle \leadsto_s \langle s_1', \sigma'\rangle}{\langle s_1 \texttt{;} \ s_2, \sigma\rangle \leadsto_s \langle s_1' \texttt{;} \ s_2, \sigma'\rangle} \qquad \frac{}{\langle \texttt{do-nothing;} \ s_2, \sigma\rangle \leadsto_s \langle s_2, \sigma\rangle}$$

$$\frac{}{\langle \texttt{while (}e\texttt{) \{ } s \texttt{ \}}, \sigma\rangle \leadsto_s \langle \texttt{if (}e\texttt{) \{ } s\texttt{; while (}e\texttt{) \{ } s \texttt{ \} \} else \{ do-nothing \}}, \sigma\rangle}$$

Математик сказал бы, что это множество *правил вывода*, которое определяет *отношение свертки* на абстрактных синтаксических деревьях SIMPLE. Ну а с точки зрения практика это нагромождение странных символов, которые не говорят ничего внятного о смысле компьютерных программ.

Мы оставим попытки разобраться в этой формальной нотации напрямую, а посмотрим, как записать те же самые правила вывода на Ruby. Ruby в качестве метаязыка понятнее программисту, и к тому же мы получаем дополнительное преимущество – возможность исполнять правила и смотреть, как они работают.

Мы *не* пытаемся описать семантику SIMPLE в виде «спецификации путем реализации». Основная причина, почему мы выбрали для описания семантики Ruby, а не математическую нотацию, – помочь разобраться в ней человеку. А то, что мы попутно получили исполняемую реализацию языка, – просто приятный бонус.

Большой недостаток использования Ruby заключается в том, что мы объясняем простой язык с использованием более сложного, что, вероятно, подрывает философские основы обучения. Следует помнить, что именно математические правила являются авторитетным описанием семантики, а Ruby лишь помогает понять, что эти правила означают.

Выражения

Начнем с семантики выражений SIMPLE. Правила оперируют абстрактными синтаксическими деревьями таких выражений, поэтому мы должны уметь представлять выражения SIMPLE в виде объектов Ruby. Сделать это можно, например, определив классы Ruby для каждого синтаксического элемента SIMPLE – чисел, операций сложения, умножения и т. д., а затем представив выражение в виде дерева, состоящего из экземпляров этих классов.

Вот, например, как выглядят определения классов `Number`, `Add` и `Multiply`:

```
class Number < Struct.new(:value)
end

class Add < Struct.new(:left, :right)
end

class Multiply < Struct.new(:left, :right)
end
```

Мы можем создать экземпляры этих классов и построить из них дерево вручную:

```
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
)
=> #<struct Add
      left=#<struct Multiply
        left=#<struct Number value=1>,
        right=#<struct Number value=2>
      >,
      right=#<struct Multiply
        left=#<struct Number value=3>,
```

```
        right=#<struct Number value=4>
      >
    >
```

Конечно, мы хотим, чтобы в конечном итоге эти деревья автоматически создавались синтаксическим анализатором. Как это делается, мы увидим в разделе «Реализация синтаксических анализаторов» ниже на стр. 82.

Классы `Number`, `Add` и `Multiply` наследуют обобщенное определение метода `#inspect` от класса `Struct`, поэтому строковые представления их экземпляров в оболочке IRB содержат много ненужных деталей. Чтобы содержимое абстрактного синтаксического дерева выглядело в IRB понятнее, мы переопределим метод `#inspect` в каждом классе[1] и будем возвращать специализированное строковое представление:

```
class Number
    def to_s
        value.to_s
    end

    def inspect
        "«#{self}»"
    end
end

class Add
    def to_s
        "#{left} + #{right}"
    end

    def inspect
        "«#{self}»"
    end
end

class Multiply
    def to_s
        "#{left} + #{right}"
    end

    def inspect
        "«#{self}»"
    end
end
```

[1] Для простоты мы воспротивились искушению вынести общий код в суперкласс или модуль.

Теперь любое абстрактное синтаксическое дерево выглядит в IRB как короткая строка, содержащая исходный код на языке SIMPLE, заключенный в «шевроны», чтобы отличить его от обычного значения в смысле Ruby:

```
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
   )
=> «1 * 2 + 3 * 4»
>> Number.new(5)
=> «5»
```

В наших примитивных реализациях метода #to_s не принимается во внимание приоритет операторов, поэтому иногда результат оказывается неправильным с точки зрения традиционных правил предшествования операций (например, приоритет * выше, чем у +). Возьмем, к примеру, такое абстрактное синтаксическое дерево:

```
>> Multiply.new(
     Number.new(1),
     Multiply.new(
       Add.new(Number.new(2), Number.new(3)),
       Number.new(4)
     )
   )
=> «1 * 2 + 3 * 4»
```

Это дерево соответствует выражению «1 * (2 + 3) * 4», отличающемуся от «1 * 2 + 3 * 4», но из строкового представления это не видно.

Проблема серьезная, но имеющая лишь косвенное отношение к теме семантики. Чтобы не усложнять изложение, мы временно проигнорируем ее и просто не будем создавать выражения, для которых получается неправильно строковое представление. А корректное решение – уже для другого языка – будет приведено в разделе «Синтаксис» на стр. 109.

Теперь можно приступить к реализации операционной семантики мелких шагов, для чего нужно будет определить методы, выполняющие свертку абстрактных синтаксических деревьев. Каждый такой метод будет принимать абстрактное синтаксическое дерево в качестве параметра, сворачивать его тем или иным способом и возвращать получившееся в результате дерево.

Но прежде чем реализовывать саму свертку, мы должны научиться отличать выражения, допускающие свертку, от не допускающих. Выражения `Add` и `Multiply` всегда можно свернуть – оба они представляют операции и могут быть преобразованы в результат этой операции путем соответствующего вычисления. Но выражение `Number` представляет значение, которое ни во что свернуть нельзя.

В принципе, различить эти два вида выражений можно было бы с помощью предиката `#reducible?`, который возвращает `true` или `false` в зависимости от класса своего аргумента:

```
def reducible?(expression)
  case expression
  when Number
    false
  when Add, Multiply
    true
  end
end
```

В Ruby при выполнении предложения `case` управляющее выражение сопоставляется с ветвями путем вызова метода `#===` значения в каждой ветви, которому в качестве аргумента передается значение управляющего выражения. Реализация метода `#===` для классовых объектов проверяет, является ли аргумент экземпляром этого класса или какого-либо его подкласса, поэтому мы можем воспользоваться синтаксической конструкцией `case object when classname` для сопоставления объекта с классом.

Однако в общем случае такой код в объектно-ориентированном языке считается дурным тоном[1]; если поведение некоторой операции зависит от класса ее аргумента, то стандартный подход состоит в том, чтобы реализовать это поведение в методе экземпляра этого класса и дать языку возможность самостоятельно решить, какой метод вызывать, а не использовать для этой цели явное предложение `case`.

Поэтому давайте напишем методы `#reducible?` в каждом из классов `Number`, `Add` и `Multiply`:

```
class Number
  def reducible?
    false
  end
```

[1] Хотя именно так мы написали бы метод `#reducible?` в функциональном языке типа Haskell или ML.

```
end

class Add
  def reducible?
    true
  end
end

class Multiply
  def reducible?
    true
  end
end
```

Это дает нужное нам поведение:

```
>> Number.new(1).reducible?
=> false
>> Add.new(Number.new(1), Number.new(2)).reducible?
=> true
```

Теперь мы можем реализовать свертку выражений; для этого точно так же определим метод `#reduce` в классах `Add` и `Multiply`. Определять метод `Number#reduce` необязательно, потому что числа не сворачиваются, так что мы должны лишь следить за тем, чтобы не вызывать `#reduce` для выражений, не допускающих свертку.

Итак, по каким правилам сворачивается выражение сложения? Если левый и правый аргумент – числа, то достаточно просто сложить их, но что, если один или оба аргумента сами нуждаются в сворачивании? Поскольку мы говорим о мелких шагах, то должны решить, какой аргумент сворачивать первым, если свертку допускают оба[1]. Обычная стратегия – сворачивать аргументы слева направо, то есть правила формулируются так.

- Если левый аргумент операции сложения допускает свертку, свернуть левый аргумент.
- Если левый аргумент операции сложения не допускает свертку, а правый допускает, свернуть правый аргумент.
- Если ни один аргумент не допускает свертку, то оба должны быть числами, поэтому сложить их.

Структура этих правил характерна для операционной семантики мелких шагов. В каждом правиле постулируется общий вид выражения, к которому оно применяется – сложение со сворачиваемым

[1] В данный момент не важно, какой именно порядок мы выберем, но вовсе избежать этого решения не удастся.

левым аргументом, со сворачиваемым правым аргументом и с двумя несворачиваемыми аргументами соответственно – и описывается, как в этом случае построить новое свернутое выражение. Выбрав данные конкретные правила, мы специфицировали, что в языке SIMPLE при вычислении выражения сложения аргументы сворачиваются слева направо, а, кроме того, определили, как объединить аргументы после выполнения свертки.

Эти правила можно непосредственно транслировать в реализацию метода `Add#reduce`, и почти так же будет выглядеть код метода `Multiply#reduce` (с тем отличием, что аргументы нужно перемножать, а не складывать).

```
class Add
  def reduce
    if left.reducible?
      Add.new(left.reduce, right)
    elsif right.reducible?
      Add.new(left, right.reduce)
    else
      Number.new(left.value + right.value)
    end
  end
end

class Multiply
  def reduce
    if left.reducible?
      Multiply.new(left.reduce, right)
    elsif right.reducible?
      Multiply.new(left, right.reduce)
    else
      Number.new(left.value + right.value)
    end
  end
end
```

Метод `#reduce` всегда строит новое выражение, а не модифицирует существующее.

Реализовав метод `#reduce` для этих видов выражений, мы можем, многократно применяя его, вычислить полное выражение путем последовательного выполнения мелких шагов.

```
>> expression =
      Add.new(
        Multiply.new(Number.new(1), Number.new(2)),
        Multiply.new(Number.new(3), Number.new(4))
      )
```

```
=> «1 * 2 + 3 * 4»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «2 + 3 * 4»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «2 + 12»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «14»
>> expression.reducible?
=> false
```

Отметим, что метод `#reduce` всегда преобразует одно выражение в другое в полном соответствии с принципом работы правил операционной семантики мелких шагов. В частности, `Add.new(Number.new(2), Number.new(12)).reduce` возвращает `Number.new(14)`, то есть представление выражения языка SIMPLE, а не 14 – число в смысле Ruby.

Такое разделение между языком SIMPLE, семантику которого мы *специфицируем*, и метаязыком Ruby, на котором *записывается спецификация*, проще поддерживать, когда различия между обоими языками абсолютно очевидны – как в случае, когда метаязыком является математический формализм, а не язык программирования. Нам же приходится быть более внимательными, потому что языки выглядят очень похоже.

Запоминая текущее выражение в качестве состояния программы и в цикле вызывая для него методы `#reducible?` и `#reduce`, пока не получится значение, мы моделируем работу абстрактной машины для вычисления выражений. Чтобы избавить себя от лишнего труда и сделать идею абстрактной машины более конкретной, мы легко можем написать на Ruby код, который делает за нас всю работу. Обернем код и состояние в класс, который назовем *виртуальной машиной*:

```
class Machine < Struct.new(:expression)
  def step
    self.expression = expression.reduce
  end

  def run
    while expression.reducible?
      puts expression
```

```
      step
    end
    puts expression
  end
end
```

Теперь мы можем создать экземпляр виртуальной машины, передав ему выражение, вызвать его метод `#run` и наблюдать, как происходит сворачивание:

```
>> Machine.new(
      Add.new(
        Multiply.new(Number.new(1), Number.new(2)),
        Multiply.new(Number.new(3), Number.new(4))
      )
    ).run
1 * 2 + 3 * 4
2 + 3 * 4
2 + 12
14
=> nil
```

Нетрудно обобщить эту реализацию на другие простые значения и операции: вычитание и деление, булевы значения `true` и `false`, логические операции `and`, `or` и `not`, операторы сравнения чисел, возвращающие булевы значения, и т. д. Вот, например, как выглядят реализации булевых значений и оператора «меньше»:

```
class Boolean < Struct.new(:value)
  def to_s
    value.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    false
  end
end

class LessThan < Struct.new(:left, :right)
  def to_s
    "#{left} < #{right}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
```

```
    true
  end

  def reduce
    if left.reducible?
      LessThan.new(left.reduce, right)
    elsif right.reducible?
      LessThan.new(left, right.reduce)
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

Как и раньше, это позволяет свернуть булево выражение мелкими шагами:

```
>> Machine.new(
      LessThan.new(Number.new(5), Add.new(Number.new(2), Number.new(2)))
    ).run
5 < 2 + 2
5 < 4
false
=> nil
```

До сих пор все было просто: мы начали специфицировать операционную семантику языка, реализовав виртуальную машину, которая может вычислять выражения этого языка. В настоящий момент состояние виртуальной машины включает всего лишь текущее выражение, а поведение описывается набором правил, которые определяют, как изменяется состояние в ходе работы машины. Мы реализовали машину в виде программы, которая запоминает текущее выражение и продолжает сворачивать его, изменяя на каждом шаге, пока еще есть возможность произвести свертку.

Но язык простых алгебраических выражений не очень интересен и не обладает многими чертами, которые мы ожидаем даже от простейшего языка программирования. Поэтому давайте сделаем на его основе что-нибудь более совершенное, больше напоминающее язык, на котором можно было бы писать полезные программы.

Первое, что бросается в глаза, – отсутствие в языке SIMPLE переменных. От любого сколько-нибудь полезного языка мы ожидаем возможности наделять значения осмысленными именами, а не работать только с литералами. Имена вводят уровень косвенности, который позволяет использовать один и тот же код для обработки различных значений, в том числе поступающих из внешнего мира и, следовательно, неизвестных во время написания программы.

Введем новый класс выражений, `Variable`, для представления переменных в Simple:

```
class Variable < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end
end
```

Чтобы свернуть переменную, абстрактная машина должна хранить соответствие между именами и значениями переменных – *окружение* – в дополнение к текущему выражению. В Ruby для хранения такого соответствия можно использовать хеш, в котором ключами являются символы, а значениями – объекты выражений. Например, хеш `{ x: Number.new(2), y: Boolean.new(false) }` представляет окружение, в котором переменные `x` и `y` ассоциированы соответственно с числом и булевым значением Simple.

В данном языке мы хотим, чтобы окружение позволяло сопоставлять имена переменных только с несвертываемыми значениями типа `Number.new(2)`, а не с произвольными допускающими свертку выражениями, например `Add.new(Number.new(1), Number.new(2))`. Мы обеспечим это ограничение позже, когда будем писать правила, изменяющие содержимое окружения.

Имея окружение, мы легко можем реализовать метод `Variable#reduce`: достаточно найти имя переменной в окружении и вернуть соответствующее значение.

```
class Variable
  def reduce(environment)
    environment[name]
  end
end
```

Отметим, что теперь мы передаем методу `#reduce` аргумент `environment`, поэтому должны изменить реализации `#reduce` в других классах выражений, так чтобы этот аргумент правильно принимался и передавался.

```
class Add
  def reduce(environment)
    if left.reducible?
      Add.new(left.reduce(environment), right)
    elsif right.reducible?
      Add.new(left, right.reduce(environment))
    else
      Number.new(left.value + right.value)
    end
  end
end

class Multiply
  def reduce(environment)
    if left.reducible?
      Multiply.new(left.reduce(environment), right)
    elsif right.reducible?
      Multiply.new(left, right.reduce(environment))
    else
      Number.new(left.value * right.value)
    end
  end
end

class LessThan
  def reduce(environment)
    if left.reducible?
      LessThan.new(left.reduce(environment), right)
    elsif right.reducible?
      LessThan.new(left, right.reduce(environment))
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

Поддержав работу с окружением во всех реализациях `#reduce`, мы должны изменить еще и саму виртуальную машину, так чтобы она запоминала окружение и передавала его методу `#reduce`:

```
Object.send(:remove_const, :Machine) # забыть старый класс Machine

class Machine < Struct.new(:expression, :environment)
  def step
    self.expression = expression.reduce(environment)
  end

  def run
    while expression.reducible?
      puts expression
      step
    end
    puts expression
  end
end
```

Метод `#run` остался прежним, но у машины появился новый атрибут `environment`, который используется в новой реализации метода `#step`.

Теперь мы можем применять свертку к выражениям, содержащим переменные, при условии, что передается окружение, в котором хранятся значения переменных:

```
>> Machine.new(
      Add.new(Variable.new(:x), Variable.new(:y)),
      { x: Number.new(3), y: Number.new(4) }
    ).run
x + y
3 + y
3 + 4
7
=> nil
```

После добавления окружения нашу операционную семантику выражений можно считать законченной. Мы спроектировали абстрактную машину, которая начинает работу с начального выражения и окружения, а затем использует текущее выражение и окружение для порождения нового выражения на каждом мелком шаге сворачивания; окружение при этом остается неизменным.

Предложения

Теперь можно рассмотреть реализацию другой программной конструкции: *предложения*. Смысл выражения в том, чтобы в результате вычисления породить другое выражение; результатом же вычисления предложения является изменение состояния абстрактной машины. Единственным состоянием нашей машины (если не считать текущего выражения) является окружение, поэтому мы разрешим предложениям языка SIMPLE порождать новое окружение, заменяющее текущее.

Простейшее из всех возможных предложений не делает вообще ничего: его нельзя свернуть, поэтому оно не может хоть как-то повлиять на окружение. Реализация тривиальна:

```
class DoNothing     ❶
  def to_s
    'do-nothing'
  end

  def inspect
    "«#{self}»"
  end
```

```
  def ==(other_statement) ❷
    other_statement.instance_of?(DoNothing)
  end

  def reducible?
    false
  end
end
```

❶ До сих пор все наши синтаксические классы наследовали классу `Struct`, однако `DoNothing` не наследует ничему. Дело в том, что у `DoNothing` нет атрибутов, а метод `Struct.new`, к сожалению, не позволяет передать пустой список имен атрибутов.

❷ Мы хотим иметь возможность сравнивать предложения на равенство. Другие синтаксические классы наследуют реализацию `#==` от `Struct`, но в `DoNothing` мы вынуждены определить ее самостоятельно.

Предложение, которое ничего не делает, на первый взгляд кажется бессмысленным, однако очень удобно иметь специальное предложение, которое представляет программу, исполнение которой успешно завершилось. Мы сделаем так, что все остальные предложения будут сворачиваться в «`do-nothing`», завершив свою работу.

Простейший пример предложения, которое делает что-то полезное, – *присваивание* вида «`x = x + 1`», но прежде чем его реализовывать, необходимо решить, как для него должны выглядеть правила свертки.

Предложение присваивания состоит из имени переменной (`x`), знака равенства и выражения («`x + 1`»). Если выражение допускает свертку, то его можно свернуть по общим правилам сворачивания выражений и породить новое предложение присваивания, содержащее свернутое выражение. Например, свертка «`x = x + 1`» в окружении, где переменная `x` имеет значение «`2`», должно дать предложение «`x = 2 + 1`», свертка которого дает предложение «`x = 3`».

Но что потом? Если выражением уже является значение, например «`3`», то мы должны просто выполнить присваивание, то есть изменить окружение, связав это значение с соответствующей переменной. Поэтому свертка предложения порождает не только новое предложение, но и новое окружение, которое иногда будет отличаться от окружения, в котором производилась свертка.

В нашей реализации для изменения окружения мы используем метод `Hash#merge`, который создает новый хеш, не изменяя старый:

```
>> old_environment = { y: Number.new(5) }
=> {:y=>«5»}
>> new_environment = old_environment.merge({ x: Number.new(3) })
=> {:y=>«5», :x=>«3»}
>> old_environment
=> {:y=>«5»}
```

Мы могли бы модифицировать текущее окружение, а не создавать новое, однако, избегая деструктивных обновлений, мы заставляем себя строить программу так, чтобы извещения о последствиях работы `#reduce` были явными и недвусмысленными. Если `#reduce` хочет изменить текущее окружение, он должен сообщить об этом, вернув измененное окружение вызывающей программе; напротив, если он не возвращает окружение, то есть уверенность, что он не произвел никаких изменений.

Это ограничение помогает подчеркнуть различие между выражениями и предложениями. В случае выражений мы передаем окружение в `#reduce` и получаем назад свернутое выражение; новое окружение не возвращается, откуда ясно, что свертка выражения не изменяет окружения. В случае предложений мы передаем `#reduce` текущее окружение и получаем назад новое окружение, а это означает, что свертка предложения может оказывать влияние на окружение. (Иными словами, структура семантики мелких шагов для языка SIMPLE показывает, что выражения в нем *чистые*, а предложения – *нечистые*.)

Итак, сворачивание «`x = 3`» в пустом окружении должно породить новое окружение `{ x: Number.new(3) }`, но мы также ожидаем, что и само предложение будет как-то свернуто, иначе наша абстрактная машина будет бесконечно присваивать переменной `x` значение `3`. Вот для этого и нужно предложение «`do-nothing`»: завершившееся присваивание сворачивается в «`do-nothing`», показывая, что свертка закончилась и получившееся новое окружение можно считать результатом присваивания.

Подведем итог, выписав правила свертки для присваивания:

- Если выражение в предложении присваивания допускает свертку, то свернуть его и получить в результате свернутое предложение присваивания и неизменившееся окружение.
- Если выражение в предложении присваивания не допускает свертку, то изменить окружение, связав это выражение с переменной в левой части присваивания, и вернуть предложение «`do-nothing`» и новое окружение.

Этой информации достаточно для реализации класса `Assign`. Единственная трудность состоит в том, что метод `Assign#reduce` должен возвращать как предложение, так и окружение, а методы в Ruby могут возвращать только один объект. Но мы можем создать иллюзию возврата двух объектов, поместив их в массив из двух элементов и вернув этот массив.

```
class Assign < Struct.new(:name, :expression)
  def to_s
    "#{name} = #{expression}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if expression.reducible?
      [Assign.new(name, expression.reduce(environment)), environment]
    else
      [DoNothing.new, environment.merge({ name => expression })]
    end
  end
end
```

Как мы и обещали, правила свертки для класса `Assign` гарантируют, что в окружение добавляются только выражения, не допускающие свертку (то есть значения).

Как и в случае выражений, мы можем вручную вычислить предложение присваивания, сворачивая его до тех пор, пока это возможно:

```
>> statement = Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
=> «x = x + 1»
>> environment = { x: Number.new(2) }
=> {:x=>«2»}
>> statement.reducible?
=> true
>> statement, environment = statement.reduce(environment)
=> [«x = 2 + 1», {:x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«x = 3», {:x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«do-nothing», {:x=>«3»}]
>> statement.reducible?
=> false
```

Это еще более трудоемкий процесс, чем ручная свертка выражений, поэтому изменим реализацию нашей виртуальной машины, научив ее обрабатывать предложения, и на каждом шаге свертки будем показывать текущее предложение и окружение:

```
Object.send(:remove_const, :Machine)

class Machine < Struct.new(:statement, :environment)
  def step
    self.statement, self.environment = statement.reduce(environment)
  end

  def run
    while statement.reducible?
      puts "#{statement}, #{environment}"
      step
    end
    puts "#{statement}, #{environment}"
  end
end
```

Теперь всю работу может сделать машина:

```
>> Machine.new(
     Assign.new(:x, Add.new(Variable.new(:x), Number.new(1))),
       { x: Number.new(2) }
     ).run
x = x + 1, {:x=>«2»}
x = 2 + 1, {:x=>«2»}
x = 3, {:x=>«2»}
do-nothing, {:x=>«3»}
=> nil
```

Как видим, машина по-прежнему выполняет шаги свертки выражений («x + 1» в «2 + 1» и затем в «3»), но теперь они производятся внутри предложения, а не на верхнем уровне синтаксического дерева.

Зная, как работает свертка предложения, мы можем обобщить этот механизм на другие виды предложений. Начнем с условного предложения вида «if (x) { y = 1 } else { y = 2 }», которое содержит выражение, называемое *условием* («x»), и два предложения, которые мы будем называть *следствием* («y = 1») и *альтернативой* («y = 2»)[1]. Правила свертки условных предложений очевидны:

[1] Это условное предложение отличается от конструкции if в Ruby. В Ruby if – выражение, которое возвращает значение, а в SIMPLE – предложение, которое выбирает, какое из двух предложений вычислять, и его единственный результат – это воздействие на текущее окружение.