Свойства оператора свёртки

Информация об авторе	
Автор:	Грэм Хаттон (Graham Hutton)
Место работы:	Университет Ноттингема, Великобритания
Электронный адрес:	gmh@cs.nott.ac.uk [1]
Адрес оригинала статьи:	Личная страница Грэма Хаттона [2]

Аннотация

В функциональном программировании fold является стандартным оператором, который инкапсулирует простой образец рекурсии для функций обработки списков. Данное руководство построено на двух ключевых аспектах оператора свёртки. Прежде всего, речь пойдёт об использовании универсального свойства оператора fold как с точки зрения способа доказательства, т.е. принципа, который не требует доказательств по индукции, так и с точки зрения способа определения, т.е. принципа, основанного на преобразовании рекурсивных функций в определения, с использованием оператора свёртки. Во-вторых, мы покажем, что даже в случае, если образец рекурсии, инкапсулированный оператором свёртки, достаточно прост, в языке с кортежами оператор имеет бóльшую степень выразительности, чем ожидалось.

Ключевые слова́: свёртка, рекурсия, доказательство свойств функций.

Введение

Многие программы, которые используют повторяющиеся операции, могут быть естественно описаны с использованием некоторой формы рекурсии, а свойства доказаны с использованием некоторой формы индукции.

Действительно, в функциональном подходе к программированию, рекурсия и индукция являются первичными инструментами для определения и доказательства свойств программ.

Не удивительно, что многие рекурсивные программы содержат общий образец рекурсии, а индуктивные доказательства содержат общий образец индукции. Повторение тех же самых операций ведёт к ошибкам, и является достаточно трудоёмким и ненужным процессом. Повторений можно избежать с помощью введения специального оператора рекурсии и способа доказательства, которые инкапсулируют общие образцы, позволяя сконцентрироваться на частях, являющихся различными для разных приложений.

В функциональном программировании оператор свёртки (также известный как foldr) — стандартный оператор, инкапсулирующий общий образец рекурсии для функций обработки списков. Оператор свёртки обладает свойством универсальности, которое инкапсулирует общий образец индуктивного доказательства, касающегося списков.

Руководство построено на двух ключевых аспектах оператора свёртки. Прежде всего, речь пойдет об универсальности оператора fold (вместе со свойством объединения), как с точки зрения способа доказательства, т.е. принципа, который не требует доказательств по индукции, так и с точки зрения способа определения, т.е. принципа, основанного на преобразовании рекурсивных функций в определения, с использованием оператора свёртки. Во-вторых, мы покажем, что даже притом, что образец рекурсии, инкапсулированный оператором свёртки, достаточно прост, в языке с кортежами оператор имеет большую степень выразительности, чем ожидалось, таким образом разрешая универсальное свойство и свойство объединения для оператора свёртки, оказывается, что он может быть применён к большему классу программ.

Статья заканчивается обзором работ, описывающих операторы рекурсии, некоторые положения которых, не будут присутствовать в данном руководстве.

Статья нацелена на читателя, знакомого с основами функционального программирования (Bird & Wadler, 1988; Bird, 1998). Все программы, приведённые в статье, написаны на языке Haskell (Peterson и др., 1997), который является стандартным функциональным языком программирования. В программах не использовались специальные возможности языка Haskell, что позволяет легко приспособить примеры к другим функциональным языкам.

Оператор свёртки

Оператор свёртки описывается в теории рекурсии (Kleene, 1952). Использование свёртки, как центрального понятия в языке программирования относится ко времени оператора редукции языка APL (Iverson, 1962), и позже оператора вставки FP (Backus, 1978). В языке Haskell, оператор свёртки для списков может быть определён следующим образом:

```
fold                        ::      (a -> b -> b) -> b -> ([a] -> b)
fold f v []             =      v
fold f v (x:xs)       =      f x (fold f v xs)
```

Таким образом, рассматривая функцию f типа a b b и значение v типа b, функция fold f v обрабатывает список типа [a], для задания значения типа b, заменяя пустой элемент nil значением v, а конструктор подстановки cons (:) в списке, функцией f. Таким же образом, оператор свёртки инкапсулирует простой образец рекурсии, в котором эти два конструктора просто заменены другими значениями и функциями. Целый ряд знакомых функций, построенных на списках, могут быть легко реализованы с использованием оператора свёртки. Например:

```
sum             ::      [Int] -> Int
sum             =      fold (+) 0

product       ::      [Int] -> Int
product       =      fold (*) 1

and             ::      [Bool] -> Bool
and             =      fold (&&) True

or             ::      [Bool] -> Bool
or             =      fold (||) Fals
```

Вспомните, что применение инфиксного оператора в круглых скобках ( ) преобразовывает оператор в префиксную форму. Этот приём, называется секционированием, и часто является полезным при написании простых функций с использованием свёртки. Если требуется, один из аргументов оператора может быть также заключён в круглые скобки. Например, функция (++) может быть реализована следующим образом:

```
(++)             ::      [a] -> [a] -> [a]
(++ ys)       =      fold (:) ys
```

Во всех приведённых примерах, конструктор (:) заменён встроенной функцией. Однако, в большинстве случаев применения оператора свёртки, конструктор (:) будет заменён функциями, определёнными пользователем, часто реализованные, как неименованные функции, с использованием абстракции, как в следующих примерах стандартных функций обработки списков:

```
length      ::      [a] -> Int
length      =      fold (\x n -> 1 + n) 0

reverse      ::      [a] -> [a]
reverse      =      fold (\x xs -> xs ++ [x]) []

map            ::      (a -> b) -> ([a] -> [b])
map f      =      fold (\x xs -> f x : xs) []

filter      ::      (a -> Bool) -> ([a] -> [a])
filter p      =      fold (\x xs -> if p x then x : xs else xs) []
```

Программы, написанные с использованием свёртки, могут быть менее удобочитаемы, чем программы, написанные с использованием явной рекурсии. Однако, программы написанные с использованием оператора fold более систематизированы, и более удобны для преобразования и доказательства различных свойств. Например, далее в статье описано, как вышеупомянутое определение функции map с использованием оператора свёртки, может быть построено из обычного определения, с использованием явной рекурсии, и что ещё более важно, как использование свёртки упрощает процесс доказательства свойств функции map.

Универсальность оператора свёртки

Как и сам оператор свёртки, свойство его универсальности происходит из теории рекурсий. Впервые этим свойством в функциональном программировании воспользовался Malcolm (1990a) в его обобщении теории списков (Bird, 1989; Meertens, 1983). Для конечных списков свойство универсальности оператора свёртки может быть задано как равенство между двумя определениями функции g для обработки списков:

```
g []            =      v
g (x:xs)      =      f x (g xs)

g            =      fold      f v
```

Подставляя справа налево в оба равенства g = fold f v, получаем рекурсивное определение для оператора fold. И, наоборот, подставляя слева направо в два уравнения для g, получаем, что g = fold f v использует прямое доказательство по индукции для конечных списков (Bird, 1998). Универсальное свойство показывает, что для конечных списков функция fold f v — не просто решение этих уравнений, а фактически их единственное решение.

Преимущество универсальности оператора свёртки в том, что его использование подтверждает два предположения, требуемых для конкретного образца доказательства по индукции. В особых случаях, когда, проверяя эти два предположения (которые могут быть сделаны без использования индукции), мы можем обратиться к универсальному свойству для завершения индуктивного доказательства того, что g = fold f v. Таким же образом универсальность оператора свёртки инкапсулирует простой образец индуктивного доказательства по отношению к спискам, так же, как и сам оператор свёртки инкапсулирует простой образец рекурсии.

Универсальное свойство свёртки может быть обобщено, для бесконечных списков и для других частных случаев, когда использование этого свойства может быть удобным (Bird, 1998). Однако, для простоты понимания, в этом руководстве мы будем рассматривать только конечные списки.

Универсальность как принцип доказательства

На основании вышесказанного, универсальное свойство оператора свёртки может выступать как принцип доказательства, при котором можно избежать индуктивных доказательств. В качестве простого примера, рассмотрим следующее уравнение между функциями, которые обрабатывают список чисел:

```
(+1) . sum      =      fold (+) 1
```

Функция слева суммирует список и затем увеличивает результат на единицу. Функция справа обрабатывает список, заменяя каждый (:) функцией (+) и пустой список [] единицей. Равенство показывает, что эти функции всегда возвращают одинаковые результаты, если их применить к одному и тому же списку.

Чтобы доказать вышеупомянутое уравнение, заметим, что оно соответствует правой части g = fold f v универсального свойства оператора свёртки, g = (+1) . sum, f = (+), и v = 1. Следовательно, пользуясь свойством универсальности оператора свёртки, получаем, что равенство эквивалентно следующим двум уравнениям:

```
((+1) . sum) []             =       1
((+1) . sum) (x:xs)         =       (+) x (((+1) . sum) xs)
```

На первый взгляд, они могут показаться более сложными, чем исходное уравнение. Однако, применение композиции и секционирования даёт:

```
sum [] +1                   =       1
sum (x:xs) +1               =       x + (sum xs + 1)
```

что может быть проверено с помощью несложных вычислений, приведённых в двух колонках:

```
sum [] + 1 = {определение sum}                  sum (x:xs) + 1 = {определение sum}
0 + 1 = {определение +}                              (x + sum xs) + 1 = {определение +}
1                                                      x + (sum xs + 1)
```

На этом доказательство заканчивается. Обычно это доказательство требовало бы явного использования индукции. Однако в нём использование индукции было инкапсулировано с помощью универсального свойства оператора свёртки, так что в итоге доказательство сведено к использованию одной операции и сопровождающих её два простых вычисления.

Вообще, любые две функции, работающие со списками, которые могут быть доказаны при помощи индукции, также могут быть доказаны с использованием универсальности оператора fold, придерживаясь того, что функции могут быть выражены с помощью оператора свёртки. Степень выразительности оператора свёртки будет рассматриваться позже в статье.

Свойство объединения оператора свёртки

Теперь позвольте нам проанализировать пример с функцией sum и рассмотреть уравнение между функциями, которые обрабатывают список значений:

```
h . fold g w      =      fold f v
```

Это уравнение часто возникает, при рассмотрении программ, написанных с использованием свёртки. Оно не верно, но мы можем использовать универсальность оператора свёртки для вычисления условий, при которых уравнение действительно будет верным. Уравнение соответствует правой части универсального свойства, откуда мы заключаем, что оно эквивалентно следующим двум уравнениям:

```
(h . fold g w) []           =       v
(h . fold g w) (x:xs)       =       f x ((h . fold g w) xs)
```

Упрощаем, пользуясь определением композиции:

```
h (fold g w [])             =       v
h (fold g w (x:xs))       =       f x (h (fold g w xs))
```

что, в свою очередь, может быть упрощено с помощью двух следующих вычислений:

```
h (fold g w [])       =       v
h w                           =       v
```

и

```
h (fold g w (x:xs))       =       f x (h (fold g w xs))
h (g x (fold g w xs))       =       f x (h (fold g w xs))
h (g x y)                           =       f x (h y)
```

Таким образом, пользуясь универсальностью оператора свёртки, мы обошлись без явного использования индукции. Мы получили два простых условия, которых достаточно для того, чтобы гарантировать (для любых конечных списков), что композиция произвольной функции и оператор свёртки могут быть объединены в один оператор свёртки. Следуя этой интерпретации, свойство назвали — свойством объединения оператора свёртки, которое может быть определено следующим образом:

```
h w                     =       v
h (g x y)               =       f x (h y)

h . fold g w       =       fold f v
```

Впервые в функциональном программировании этим свойством воспользовался Malcolm (1990a) в обобщении более ранних работ (Bird, 1989; Meertens, 1983). Как и свойство универсальности, свойство объединения — способ доказательства, не требующий индукции. Фактически, для большинства практических примеров это свойство чаще оказывается более предпочтительным, чем свойство универсальности. В качестве примера снова рассмотрим уже знакомое равенство:

```
(+1) . sum       =       fold (+) 1
```

В предыдущем пункте это уравнение было доказано использованием свойства универсальности. Однако, при использовании свойства объединения оператора свёртки, полученное доказательство оказывается более простым. Прежде всего, мы заменяем функцию sum определением с использованием свёртки, данного ранее:

```
(+1) . fold (+) 0             =       fold (+) 1
```

Теперь уравнение можно преобразовать с использованием свойства объединения. Получаем, что уравнение следует из следующих двух предположений:

```
(+1) 0                    =       1
(+1) ((+) x y)              =       (+) x ((+1) y)
```

Упрощение этих уравнений, пользуясь определением и секционированием, даёт, 0 + 1 = 1 и (x + y) + 1 = x + (y + 1). Полученные равенства являются арифметически верными. В более обобщённом виде добавляем в этом примере произвольный инфиксный оператор ⊕, который является ассоциативным. Применив свойство объединения, получаем, что:

```
(⊕ a) . fold (⊕) b             =       fold (⊕) (b ⊕ a)
```

Более интересным примером является следующее хорошо известное уравнение, которое показывает, что оператор map использует композицию (.):

```
map f . map g            =      map (f . g)
```

Заменяя второе и третье вхождение оператора map в уравнении его определением с использованием свёртки, получаем, что равенство может быть переписано в виде, подходящем для применения свойства объединения:

```
map f . fold (\x xs -> g x : xs) []

fold (\x xs -> (f . g) x : xs) []
```

Обращаясь к свойству объединения, упрощаем. Получаем следующие два уравнения, которые являются верными по определению map и (.):

```
map f []                  =      []
map f (g x : y)      =      (f . g) x : map f y
```

В дополнение к вышеупомянутым особенностям, есть множество других полезных свойств оператора свёртки, которые могут быть получены при помощи свойства универсальности оператора свёртки (Bird, 1998). Однако, свойство объединения применяется в большинстве практических случаев, и всегда может быть приведено к свойству универсальности, если его невозможно применить.

Универсальность как способ определения

Будучи используемым в качестве принципа доказательства, универсальное свойство оператора свёртки может быть использовано как способ определения, который ведёт преобразование рекурсивных функций в соответствии с определением, используя свёртку. В качестве примера рассмотрим рекурсивно определенную функцию sum, которая вычисляет сумму списка чисел:

```
sum                  ::       [Int] -> Int
sum []             =       0
sum (x:xs)       =       x + sum xs
```

Предположим теперь, что мы хотим переопределить эту функцию, используя свёртку. Таким образом, мы хотим решить уравнение sum = fold f v для функции f и значения v. Заметим, что уравнение соответствует правой части универсального свойства оператора свёртки. Отсюда получаем, что уравнение эквивалентно следующим двум равенствам:

```
sum []             =       v
sum (x:xs)       =       f x (sum xs)
```

С использованием первого уравнения и определения sum, сразу же получаем, что v = 0. Из второго уравнения мы вычисляем определение для f следующим образом:

```
sum (x:xs)       =       f x (sum xs)
x + sum xs       =       f x (sum xs)
x + y             =       f x y
f                    =       (+)
```

Пользуясь универсальностью оператора свёртки, получаем:

```
sum                  =       fold (+) 0
```

Отметим, что ключевым моментом в вычислении определения для f является обобщение выражения sum xs для новой переменной y. Фактически, такой шаг обобщения не является специфичным для функции sum, но этот шаг будет ключевым в преобразовании какой-либо рекурсивной функции из определения с использованием оператора свёртки.

Конечно, пример с функцией sum — достаточно искусственный, потому что её определение непосредственно использует оператор свёртки. Однако, есть много примеров функций, чьё определение использует свёртку не настолько явно. Например, рассмотрим рекурсивно определенную функцию map f, которая применяет функцию f к каждому элементу списка:

```
map                   ::       (a -> b) -> ([a] -> [b])
map f []              =        []
map f (x:xs)          =        f x : map f xs
```

Для переопределения функции map f, использующей свёртку, мы должны решить уравнение map f = fold g v для функции g и значения v. Используя универсальность оператора свёртки, получаем, что это уравнение эквивалентно следующим двум уравнениям:

```
map f []              =        v
map f (x:xs)          =        g x (map f xs)
```

Из первого уравнения по определению функции map непосредственно получаем, что v = []. По второму уравнению, вычисляем определение для g следующим образом:

```
map f (x:xs)              =       g x (map f xs)
f x : map f xs              =       g x (map f xs)
f x : ys                    =       g x ys
g                           =       \x ys -> f x :ys
```

Таким образом, с использованием универсальности оператора свёртки, мы вычислили что:

```
map f                   =       fold (\x ys -> f x : ys) []
```

Вообще, любая функция, работающая со списками, и которая может быть выражена с использованием оператора свёртки, может быть преобразована к такому виду при помощи свойства универсальности оператора свёртки.

Увеличение степени свёртки: производство кортежей

Начнем с примера использования свёртки для создания кортежей рассмотрим функцию sumlength, которая возвращает сумму и длину списка чисел:

```
sumlength             ::       [Int] -> (Int, Int)
sumlength             =       (sum xs, length xs)
```

С помощью объединения определений функций sum и length с использованием оператора свёртки, описанная ранее функция sumlength, возвращающая пару чисел из списка чисел, может быть определена с использованием свёртки:

```
sumlength             =       fold (\n (x, y) -> (n + x, 1 + y)) (0, 0)
```

Это определение более эффективно, чем первое, т.к. обрабатываемый список просматривается один раз. На основании этого примера получаем, что любое двойное применение свёртки к одному и тому же списку, может быть всегда заменено единственным использованием свёртки, который генерирует пару, с использованием так называемого свойства «бананового соединения» («banana split» property) оператора свёртки (Meijer, 1992).

Такое странное название этого свойства появилось из-за того, что оператор свёртки иногда записывается с использованием скобок (| |), которые напоминают бананы, а соединяющий их оператор иногда называют соединением. Следовательно, их комбинацию можно назвать банановым соединением!

Приведём более интересный пример, рассмотрим функцию dropWhile p, которая удаляет начальные элементы из списка, пока они удовлетворяют предикату p:

```
dropWhile                          ::       (a -> Bool) -> ([a] -> [a])
dropWhile p []                    =       []
dropWhile p (x:xs)              =       if p x then dropWhile p xs else x:xs
```

Предположим, что мы хотим переопределить эту функцию с использованием оператора свёртки. Используя универсальность оператора, заключаем, что равенство dropWhile p = fold f v эквивалентно следующим двум уравнениям:

```
dropWhile p []                    =       v
dropWhile p (x:xs)              =       f x (dropWhile p xs)
```

Из первого уравнения получаем, что v = []. Из второго уравнения мы попытаемся вывести определение для f следующим образом:

```
dropWhile p (x:xs)                                      =       f x (dropWhile p xs)
if p x then dropWhile p xs else x:xs)             =       f x (dropWhile p xs)
if p x then yx else x:xs                            =       f x ys
```

К сожалению, последняя строка не является правильным определением для f, потому что переменная xs оказывается свободной. Оказывается, не возможно определить функцию dropWhile p, непосредственно используя, оператор свёртки, однако возможно косвенно, если ввести дополнительную функцию, связывающую эту переменную:

```
dropWhile'                    ::       (a -> Bool) -> ([a] -> ([a], [a]))
dropWhile'p xs              =       (dropWhile p xs, xs)
```

Она разбивает на пары результат применения dropWhile p к списку, и может быть переопределена с использованием свёртки. Применяя свойство универсальности, получаем, что равенство dropWhile' p = fold f v эквивалентно следующим двум уравнениям:

```
dropWhile'p []              =       v
dropWhile'p (x:xs)       =       f x (dropWhile'p xs)
```

Из первого уравнения получаем, что v = ([], []). Из второго уравнения мы попытаемся вывести определение для f следующим образом:

```
dropWhile'p (x:xs)                                      =       f x (dropWhile'p xs)
(dropWhile p (x:xs), x:xs)                              =       f x (dropwhile p xs, xs)
(if p x then dropWhile p xs else x:xs, x:xs)      =       f x (dropWhile p xs, xs)
(if p x then ys else x:xs, x:xs)                        =       f x (ys, xs)
```

Заметьте, что определение f, написанное в последней строке — корректно, т.к. все переменные связаны. В итоге получаем, что с использованием универсальности оператора свёртки мы получили что:

```
dropWhile'p       =       fold f v
                          where
                                f x (ys, xs)       =       (if p x then ys else x:xs, x:xs)
                                v                   =       ([], [])
```

Это определение удовлетворяет уравнению dropWhile' p xs = (dropWhile p xs, xs), но не использует dropWhile. Следовательно, функция dropWhile непосредственно может быть переопределена просто, как dropWhile p = fst . dropWhile' p.

В заключение, для функции dropWhile', возвращающей пару из желательного результата и списка параметров, мы показали, что функция dropWhile может быть переопределена в терминах свёртки, как и требовалось в примере. На самом деле полученный результат является общей теоремой (Meertens, 1992), которая говорит о том, что любая функция, обрабатывающая конечные списки и составляющая пару из желаемого результата и параметров списка, всегда может быть переопределена в терминах свёртки, однако, это не всегда бывает выгодно. Оригинальное определение (возможно рекурсивное) для функции может оказаться более удобным.

Примитивная рекурсия

В этом пункте мы покажем, что при использовании техники составления кортежей (см. п. 4) каждая функция, определённая с использованием примитивной рекурсии и обрабатывающая списки, может быть переопределена в терминах свёртки. Вспомним, что оператор свёртки инкапсулирует простой образец рекурсии, и применим это для определения следующей функции h, который обрабатывает списки:

```
h []                  =      v
h (x:xs)              =      g x (h xs)
```

Такие функции могут быть переопределены с использованием равенства h = fold g v. Мы обобщим этот образец рекурсии до примитивной рекурсии в двух шагах. Прежде всего, вводим дополнительный аргумент y для функции h, который будет обрабатываться новой функцией f, и в случае рекурсии, оставляем неизменными функции g и h. Таким образом, рассмотрим следующий образец рекурсии для определения функции h:

```
h y []            =      f y
h y (x:xs)        =      g y x (h y xs)
```

Простым наблюдением, или с помощью обычного применения универсальности свёртки, функция h y может быть переопределена следующим образом:

```
h y                   =      fold (g y) (f y)
```

На втором шаге мы вводим список xs, как дополнительный аргумент к вспомогательной функции g. Таким образом, мы рассматриваем следующий образец для определения h:

```
h y []            =      f y
h y (x:xs)        =      g y x xs (h y xs)
```

Этот образец рекурсии называют примитивной рекурсией (Kleene, 1952). Технически, стандартное определение примитивной рекурсии требует, чтобы y был конечной последовательностью аргументов. Однако, в Haskell, рассматривая случай единственного аргумента, y оказывается вполне подходящим.

Чтобы переопределить примитивно рекурсивные функции в терминах свёртки, мы должны решить уравнение h y = fold i j для функции i и значения j. Непосредственно решить его невозможно, однако, возможно решить косвенно при помощи введения более общей функции:

```
k y xs            =      (h y xs, xs)
```

которая разбивает на пары результат применения h y к списку с самим списком, и может быть переопределена с использование свёртки. Воспользовавшись универсальностью оператора свёртки, получаем, что равенство k y = fold i j эквивалентно следующим двум уравнениям:

```
k y []            =      j
k y (x:xs)       =      i x (k y xs)
```

Из первого уравнения получаем, что j = (f y, []). Из второго уравнения, мы попытаемся вывести определение для i следующим образом:

```
k y (x:xs)                          =      i x (k y xs)
(h y (x:xs), x:xs)                    =      i x (h y xs, xs)
(g y x xs (h y xs), x:xs      =      i x (h y xs, xs)
(g y x xs z, x:xs)                    =      i x (z, xs)
```

В заключении, используя универсальность свёртки, получаем что:

```
k y             =      fold i j
                   where
                         i x (z, xs)      =      (g y x xs z, x:xs)
                         j                  =      (f y, [])
```

Это определение удовлетворяет уравнению k y xs = (h y xs, xs), но не использует h. Следовательно, примитивно-рекурсивную функцию h можно переопределить с использованием равенства h y = fst . k y. Мы показали, как произвольная примитивно-рекурсивная функция, обрабатывающая списки, может быть переопределена в терминах свёртки.

Заметьте, что использование способа составления кортежей для определения примитивно-рекурсивных функций в терминах свёртки, является ключевым моментом к определению функции предшественника для нумералов Чёрча (Barendregt, 1984). Действительно, интуитивное представление естественного числа (или вообще, любого индуктивного типа данных) в исчислении заключается в представлении каждого числа его оператором свёртки. Например, число 3 = succ (succ (succ zero)) представляется термом \f x -> f (f (f x)), который является оператором свёртки для числа 3, если рассматривать f и x, как succ и zero соответственно.

Использование оператора свёртки для определения функций

Степень примитивной рекурсии может увеличиваться, а, следовательно, и степень оператора свёртки также может увеличиваться. Приведём ещё один пример использования свёртки, рассмотрим функцию compose, которая формирует список функций. Она может быть определена с использованием свёртки, путём замены каждого конструктора (:) в списке, функцией (.), а пустой список [], идентично работающей функцией id:

```
compose      ::      [a -> a] -> (a -> a)
compose      =      fold (.) id
```

Рассмотрим проблему суммирования списка чисел. Основным определением для этой функции является: sum = fold (+) 0, числа в списке рассматриваются справа налево. Можно также реализовать функцию suml, которая будет обрабатывать список в обратном порядке. Функция suml задаётся с использованием вспомогательной функции suml', которая определяется через явную рекурсию и использует накапливающий параметр n:

```
suml            ::      [Int] -> Int
suml xs      =      suml'xs 0
                   where
                         suml'[] n            =      n
                         suml'(x:xs) n            =      suml'xs (n + x)
```

Поскольку функция (+) ассоциативна, она возвращает тот же самый результат при обработке одного и того же списка. Однако применение функции suml оказывается более рациональным из-за того, что её можно легко

модифицировать (Bird, 1998).

Теперь предположим, что мы хотим переопределить функцию suml с использованием оператора свёртки. Этого можно добиться только воспользовавшись вспомогательной функцией:

```
suml'                   ::      [Int] -> (Int -> Int)
```

которая может переопределяться напрямую. Применяя свойство универсальности оператора свёртки, получаем, что равенство suml' = fold f v эквивалентно следующим двум уравнениям:

```
suml' []                  =       v
suml' (x:xs)              =       f x (suml' xs)
```

Из первого уравнения получаем, что v = id. Из второго уравнения мы попытаемся вывести определение для f следующим образом:

```
suml' (x:xs)              =       f x (suml' xs)
suml' (x:xs) n              =       f x (suml' xs) n
suml' xs (n + x)        =       f x (suml' xs) n
g (n + x)                   =       f x g n
f                           =       \x g -> (\n -> g (n + x))
```

Теперь, применяя свойство универсальности функции fold, получаем:

```
suml'                   =       fold (\x g -> (\n -> g (n + x))) id
```

Полученное определение показывает, что suml' обрабатывает список, заменяя пустой список [] функцией id, а каждый конструктор (:) функцией, которая берет число x и функцию g, и возвращает функцию, которая берёт значение n из аккумулятора и возвращает результат применения g к новому значению аккумулятора n + x.

Заметьте, что задание функции, как suml' :: [Int] -> (Int -> Int) невозможно при определении её с использованием свёртки. В частности, если эти два аргумента реверсированы или они представляют собой пару, в этом случае тип suml' показывает, что определение больше не может быть реализовано непосредственно с использованием свёртки. На самом деле, нужно быть достаточно осторожными при переопределении функции с помощью свёртки. На первый взгляд можно подумать, что оператор свёртки используется только для тех функций, которые обрабатывают списки справа налево. Однако, на примере функции suml' мы показали, что порядок обработки элементов списка зависит от аргументов оператора, а не от самого оператора свёртки.

Мы показали, что функция suml может быть переопределена с использованием свёртки, как и требовалось:

```
suml xs                 =       fold (\x g -> (\n -> g (n + x))) id xs 0
```

В заключении отметим, что использование свёртки для составления функций, является удобным способом для применения атрибутивных грамматик в функциональных языках (Fokkinga и др., 1991; Swierstra и др., 1998).

Оператор foldl

Рассмотрим стандартный оператор foldl на примере функции suml, описанной в предыдущем разделе. Оператор обрабатывает элементы списка слева направо с использованием функции f (для объединения значений), и значение v в качестве начального условия:

```
foldl                   ::      (b -> a -> b) -> b -> ([a] -> b)
foldl f v []             =      v
foldl f v (x:xs)        =      foldl f (f v x) xs
```

С помощью этого оператора функция suml может быть переопределена просто как suml = foldl (+) 0. Наряду с suml многие другие функции быть определены с использованием этого оператора. Например, функция reverse переопределяется следующим образом:

```
reverse       ::       [a] -> [a]
reverse       =       foldl (\xs x -> x:xs) []
```

Это определение предпочтительнее, чем его стандартная форма (с использованием свёртки), т.к. в нём не требуется многократного применения оператора (++).

Простое обобщение вычислений, проведенных ранее для функции suml, показывает, что foldl выражается через оператор свёртки следующим образом:

```
foldl f v xs       =       fold (\x g -> (\a -> g (f a x))) id xs v
```

Однако, выразить оператор свёртки через foldl не возможно вследствие того, что foldl определяется строго в хвосте списка своих аргументов, а оператор свёртки — нет. Существует ряд двойственных теорем, связанных с операторами fold и foldl, а также рекомендации, помогающие определить, какой оператор лучше всего подходит для данной конкретной задачи (Bird, 1998).

Функция Акермана

В заключение рассмотрим пример реализации функции ack, иллюстрирующий степень свёртки. Функция ack обрабатывает два списка целых чисел и определяется с использованием явной рекурсии следующим образом:

```
ack                         ::      [Int] -> ([Int] -> [Int])
ack [] ys                   =       1:ys
ack (x:xs) []               =       ack xs [1]
ack (x:xs) (y:ys)       =       ack xs (ack (x:xs) ys)
```

Функция Акермана позволяет оперировать списками, а не числами, представляя каждое число n списком с n произвольных элементов. Эта функция — классический пример функции, которая в языке программирования низкого уровня не является примитивно-рекурсивной. Однако, на языке высокого уровня, типа Haskell, функция Акермана действительно примитивно-рекурсивная (Reynolds, 1985).

Для того, чтобы выразить эту функцию, используя оператор свёртки, прежде всего необходимо воспользоваться свойством универсальности оператора fold. Получаем, что равенство ack = fold f v эквивалентно следующим двум уравнениям:

```
ack []                   =       v
ack (x:xs)              =       f x (ack xs)
```

Из первого уравнения получаем, что v = (1 :). Из второго уравнения мы не сможем получить определение для f, как мы делали это раньше. Необходимо переопределить с использованием свёртки функцию ack (x:xs), стоящую в левой части второго уравнения. С использованием универсального свойства получаем следующий

результат для уравнения:

```
ack (x:xs) []            =      w
ack (x:xs) (y:ys)      =      g y (ack (x:xs) ys)
```

Из первого уравнения получаем, что w = ack xs [1], из второго уравнения получаем:

```
ack (x:xs) (y:ys)            =      g y (ack (x:xs) ys)
ack xs (ack (x:xs) ys)      =      g y (ack (x:xs) ys)
ack xs zs                          =      g y zs
g                                  =      \y -> ack xs
```

Таким образом, используя универсальность оператора свёртки, мы вычислили что:

```
ack (x:xs)                   =      fold (\y -> ack xs) (ack xs [1])
```

На основе полученного результата мы можем вычислить определение для f:

```
ack (x:xs)                                    =      f x (ack xs)
fold (\y -> ack xs) (ack xs [1])            =      f x (ack xs)
fold (\y -> g) (g [1])                        =      f x g
f                                                =      \x g -> fold (\y -> g) (g [1])
```

В заключении дважды применяем универсальное свойство оператора свёртки и получаем что:

```
ack            =      fold (\x g -> fold (\y -> g) (g [1])) (1:)
```

Другие работы, посвящённые операторам рекурсии

В последнем разделе мы кратко рассмотрим несколько работ, в которых рассматриваются операторы рекурсии, не упоминавшиеся в этом руководстве.

Оператор свёртки для регулярных типов данных. Оператор свёртки может применяться не только к спискам, но и к «регулярным» типам данных (Malcolm, 1990b; Meijer и др., 1991; Sheard & Fegaras, 1993).

Оператор свёртки для вложенных типов данных. Оператор свёртки может применяться к «вложенным» типам данных. Однако получающийся оператор не является достаточно обобщённым, чтобы было возможно его повсеместное использование. Поиск решений этой проблемы — предмет настоящего исследования (Bird & Meertens, 1998; Jones & Blampied, 1998).

Оператор свёртки для функциональных типов данных. С обобщением оператора свёртки для типов данных появляется множество технических проблем. Пользуясь идеями из теории категории, оператор свёртки может быть определён для нескольких типов данных (Meijer & Hutton, 1995), но его использование практически не востребовано. Однако, более простое, но менее общее решение положило начало использованию оператора, связанному с циклическими структурами (Fegaras & Sheard, 1996).

Оператор свёртки для монад. В ряде важных статей Wadler показал, как функциональные программы могут быть промоделированы с использованием монад (Wadler, 1990; Wadler, 1992a; Wadler, 1992b). Работа включает в себя рассмотрение способов применения оператора свёртки к структурам, обрабатывающим рекурсивные значения с использованием монад (Fokkinga, 1994; Meijer & Jeuring, 1995).

Относительная свёртка. Оператор свёртки может быть применён также к отношениям. Это обобщение позволяет добавить описательный аспект оператору свёртки к уже имеющемуся программному. Например, относительная свёртка используется для расчета схем Ruby (Jones & Sheeran, 1990; Jones, 1990), эйндховенского исчисления спекуляции (Aarts и др., 1992) и, в недавно опубликованном учебнике по алгебре программирования (Bird & de Moor, 1997).

Другие операторы рекурсии. Оператор свёртки — не единственный полезный оператор рекурсии. Например, двойственный оператор используется в целях спецификации (Jones, 1990; Bird & de Moor, 1997), для программирования реактивных систем (Kieburtz, 1998).

Автоматическое изменение программы. При написании программы с использованием операторов рекурсии можно упростить процесс оптимизации во время компиляции. Например, процедура устранения использования промежуточных структур данных в программах в значительной степени упрощается, когда программы написаны с использованием операторов рекурсии (Wadler, 1981; Launchbury & Sheard, 1995; Takano & Meijer, 1995). Основополагающая система для преобразования программ, написанных с использованием операторов рекурсии, в настоящий момент активно развивается (de Moor & Sittampalan, 1998).

Полиморфизм в программировании. Написание программ, имеющих специфический тип, положило начало новому направлению, названному полиморфизмом в программировании (Backhouseet и др., 1998). Формально такие программы задаются одним или более типом данных. Программному полиморфизму уже найдено множество применений, включая выбор подходящего образца (Jeuring, 1995), и решение различных проблем оптимизации (Bird & de Moor, 1997).

Языки программирования. Целый ряд экспериментальных языков программирования показывают, что использование операторов рекурсии является более рациональным способом задания рекурсивных функций. Сюда входят такие языки, как алгебраический язык ADL (Kieburtz & Lewis, 1994), язык программирования Charity (Cockett & Fukushima, 1992) и язык программирования PolyP (Jansson & Jeuring, 1997).

Благодарности

Я хотел бы выразить благодарность Эрику Мейджеру и членам группы Languages and Programming group в университете Ноттингема за многие часы интересных разговоров по поводу оператора fold. Я также благодарен Роланду Бакхаусу, Марку П. Джоунсу, Филиппу Уодлеру, и анониму JFP за их детальные комментарии к статье, на основании которых были проведены существенные улучшения содержания. Работа была написана при поддержке Engineering and Physical Sciences Research Council (EPSRC). Грант на исследование GR/L74491, Structured Recursive Programming.

Список литературы

1. Aarts, Chritiene, Backhouse, Roland, Hoogendijk, Paul, Voermans, Ed, & van der Woude, Jaap. (1992). A relational theory of datatypes. [3].
2. Backhouse, Roland, Jansson, Patrik, Jeuring, Johan, & Meertens, Lambert. 1998 (Sept.). Generic programming: An introduction. Lecture Notes of the 3rd International Summer School on Advanced Functional Programming.
3. Backus, John. (1978). Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. Cacm, 9 (Aug.).
4. Barendregt, Henk. (1984). The lambda calculus { it's syntax and semantics. North-Holland. Revised edition.
5. Bird, Richard. (1989). Constructive functional programming. Proc. Marktoberdorf International Summer School on Constructive Methods in Computer Science. Springer-Verlag.
6. Bird, Richard. (1998). Introduction to functional programming using Haskell (second edition). Prentice Hall.
7. Bird, Richard, & de Moor, Oege. (1997). Algebra of programming. Prentice Hall.
8. Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. Jeuring, Johan (ed), Proc. Fourth International Conference on Mathematics of Program Construction. LNCS, vol. 1422. Springer-Verlag.
9. Bird, Richard, & Wadler, Philip. (1988). An introduction to functional programming. Prentice Hall.
10. Cockett, Robin, & Fukushima, Tom. (1992). About Charity. Yellow Series Report No. 92/480/18. Department of Computer Science, The University of Calgary.

11. de Moor, Oege, & Sittampalan, Ganesh. 1998 (Sept.). Generic program transformation. Lecture Notes of the 3rd International Summer School on Advanced Functional Programming.
12. Fegaras, Leonidas, & Sheard, Tim. (1996). Revisiting catemorphisms over datatypes with embedded functions. Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.
13. Fokkinga, Maarten. 1994 (June). Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28. University of Twente.
14. Fokkinga, Maarten, Jeuring, Johan, Meertens, Lambert, & Meijer, Erik. (1991). Translating attribute grammars into catamorphisms. The Squiggolist, 2(1).
15. Hutton, Graham. (1998). Fold and unfold for program semantics. Proc. 3rd ACM SIGPLAN International Conference on Functional Programming.
16. Iverson, Kenneth E. (1962). A Programming Language. Wiley, New York.
17. Jansson, Patrick, & Jeuring, Johan. (1998). Polytypic unification. To appear in the Journal of Functional Programming.
18. Jansson, Patrik, & Jeuring, Johan. (1997). PolyP - a polytypic programming language extension. Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM Press.
19. Jeuring, Johan. (1995). Polytypic pattern matching. Proc. 7th International Conference on Functional Programming and Computer Architecture. ACM Press, San Diego, California.
20. Jones, Geraint. (1990). Designing circuits by calculation. Technical Report PRG-TR-10-90. Oxford University.
21. Jones, Geraint, & Sheeran, Mary. (1990). Circuit design in Ruby. Staunstrup (ed), Formal methods for VLSI design. Elsevier Science Publications, Amsterdam. A tutorial on the universality and expressiveness of fold 17.
22. Jones, Mark P., & Blampied, Paul. (1998). A pragmatic approach to maps and folds for parameterized datatypes. Submitted for publication.
23. Kieburtz, Richard B. (1998). Reactive functional programming. Proc. PROCOMET. Chapman and Hall.
24. Kieburtz, Richard B. & Lewis, Jeffrey. (1994). Algebraic Design Language (preliminary definition). Oregon Graduate Institute of Science and Technology.
25. Kleene, S.C. (1952). Introduction to metamathematics. Van Nostrand. Launchbury, John, & Sheard, Tim. (1995). Warm fusion: Deriving build-catas from recursive definitions. Proc. 7th International Conference on Functional Programming and Computer Architecture. ACM Press, San Diego, California.
26. Malcolm, Grant. (1990a). Algebraic data types and program transformation. Ph.D. thesis, Groningen University.
27. Malcolm, Grant. (1990b). Algebraic data types and program transformation. Science of Computer Programming, 14(2-3), 255-280.
28. Meertens, Lambert. 1983 (Nov.). Algorithmics: Towards programming as a mathematical activity. Proc. cwi symposium.
29. Meertens, Lambert. (1992). Paramorphisms. Formal Aspects of Computing, 4(5), 413-425.
30. Meijer, Erik. (1992). Calculating compilers. Ph.D. thesis, Nijmegen University.
31. Meijer, Erik, & Hutton, Graham. (1995). Bananas in space: Extending fold and unfold to exponential types. Proc. 7th International Conference on Functional Programming and Computer Architecture. ACM Press, San Diego, California.
32. Meijer, Erik, & Jeuring, Johan. (1995). Merging monads and folds for functional programming. Jeuring, Johan, & Meijer, Erik (eds), Advanced Functional Programming. LCNS, vol. 925. Springer-Verlag.
33. Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. Hughes, John (ed), Proc. Conference on Functional Programming and Computer Architecture. LNCS, no. 523. Springer-Verlag.
34. Peterson, John, et al. 1997 (Apr.). The Haskell language report, version 1.4. Available on the World-Wide-Web from [4].
35. Reynolds, John C. (1985). Three approaches to type structure. Proc. International Joint Conference on Theory and Practice of Software Development. Lecture Notes in Computer Science, vol. 185. Springer.

36. Sheard, Tim, & Fegaras, Leonidas. (1993). A fold for all seasons. Proc. ACM Conference on Functional Programming and Computer Architecture. Springer.
37. Swierstra, S. Doaitse, Alcocer, Pablo R. Azero, & Saraiva, Joao. 1998 (Sept.). Designing and implementing combinator languages. Lecture Notes of the 3rd International Summer School on Advanced Functional Programming.
38. Takano, Akihiko, & Meijer, Erik. (1995). Shortcut deforestation in calculational form. Proc. 7th International Conference on Functional Programming and Computer Architecture. ACM Press, San Diego, California.
39. Wadler, Philip. 1981 (Oct.). Applicative style programming, program transformation, and list operators. Proc. ACM Conference on Functional Programming Languages and Computer Architecture.
40. Wadler, Philip. (1990). Comprehending monads. Proc. ACM Conference on Lisp and Functional Programming.
41. Wadler, Philip. (1992a). The essence of functional programming. Proc. Principles of Programming Languages.
42. Wadler, Philip. (1992b). Monads for functional programming. Broy, Manfred (ed), Proc. Marktoberdorf Summer School on Program Design Calculi. Springer-Verlag.

Примечания

[1] mailto:gmh@cs.nott.ac.uk
[2] http://www.cs.nott.ac.uk/~gmh
[3] http://www.cs.nott.ac.uk/~rcb/MPC/book.ps.gz
[4] http://www.haskell.org

Источники и основные авторы

Свойства оператора свёртки *Источник*: http://ru.wikibooks.org/w/index.php?oldid=45599 *Редакторы:* Artem M. Pelenitsyn, Dark Magus, Exlevan, 3 анонимных правок

Лицензия