Функциональное программирование при помощи бананов, линз, конвертов и колючей проволоки

Эрик Мейер * Маартен Фоккинга † Росс Патерсон ‡

Аннотация

Мы разрабатываем исчисление для ленивого функционального программирования, основанное на связанных с определениями типов данных рекурсивных операторах. Для таких операторов мы выводим различные алгебраические законы, полезные при выводе и преобразовании программ. Мы покажем, что все функции–примеры из "Введения в функциональное программирование" Бёрда (Bird) и Уодлера (Wadler) могут быть выражены при помощи этих операторов.

1 Введение

Среди множества стилей и методологий конструирования компьютерных программ, по нашему мнению, стиль Squiggol заслуживает внимания общественности фукционального программирования. Общей целью стиля Squiggol является *вычисление* программ из их описания таким же способом, как математик находит решение дифференциальных уравнений или использует арифметику для решения численных задач.

Нетрудно констатировать, доказать и использовать законы для хорошо известных операций, таких как сложение, умножение и — на уровне функций — композиция. Однако, довольно сложно констатировать, доказать и использовать законы для произвольных рекурсивных функций, главным образом потому, что трудно рассматривать рекурсивный механизм отдельно от контекста. Алгоритмическая структура скрыта за использованием неструктурированных рекурсивных определений. Мы раскусили эту задачу, представляя различные рекурсивные схемы как отдельные функции высшего порядка и предоставляя каждой запись, независимую от компонентов, вместе с которыми такая схема составляет рекурсивно определенную функцию.

Эта философия в некотором роде похожа на методологию "структурного программирования" для императивного программирования. Использвоание произвольных goto запрещается в пользу структурных примитивов потока управления, таких как условия или циклы "пока", заменяющих жесткие шаблоны goto так, что исследование программ становится возможным и иногда даже приятным занятием. Для функциональных программ вопрос заключается в том, какие рекурсивные схемы должны быть выбраны в качестве базиса для исчисления программ. Мы рассмотрим несколько рекурсивных операторов, естественным образом связанных с определениями алгебраичесикх типов. Некоторые общие теоремы об этих

*University of Nijmegen, Department of Informatics, Toernooiveld 6525 ED Nijmegen, e-mail: eric@cs.kun.nl
†CWI, Amsterdam & University of Twente
‡Imperial College, London

операторах также доказаны и впоследствии использованы для преобразования программ и доказательства их корректности.

Бёрд (Bird) и Меертенс (Meertens) [4, 18] вывели несколько законов для специфичных типов данных (особенно для *конечных* списков), при помощи которых они получили решения различных задач программирования. Если внести это исчисление в рамки теории категорий, работа Бёрда и Меертенса о списках может быть распространена на произвольные индуктивно определенные типы данных [17, 12]. Недавно группа Бэкхауса (Backhouse) [1] обобщила исчисление на реляционные системы, таким образом охватывая неопределенность.

Независимо Патерсон (Paterson) [21] разработал исчисление функциональных программ, похожее по содержанию, но весьма отличное внешне (как многие австралийские животные) от работ, на которые мы ссылались выше. И действительно, если пройтись по синтаксическим различиям, законы, выведенные Патерсоном, такие же, и в некоторых случаях немного более общие, нежели разработанные Squiggol'ерами.

Эта статья расширяет теорию на контекст ленивого функционального программирования, т.е. для нас тип — ω-cpo, и мы рассматриваем только непрерывные функции между типами (с точки зрения теории категорий, мы работаем в категории CPO). Работа в категории SET, как она проделана, например, Малькомом (Malcom) [17] или Хагино (Hagino) [14], подразумевает, что конечные типы данных (определенные как начальные алгебры) и бесконечные типы данных (определенные как терминальные коалгебры) — две разные вещи. В таком случае невозможно определить при помощи индукции функции (катаморфизмы), которые были бы применимы к конечным и бесконечным типам данных одновременно, и произвольные рекурсивные определения не могут быть использованы. Работа в CPO дает преимущество в том, что носители начальных алгебр и терминальных коалгебр совпадают, таким образом, существует единый тип данных, охватывающий и конечные, и бесконечные элементы. Однако, за это придется расплачиваться тем, что частичная применимость и функций, и значений становится неизбежной.

2 Списки

Мы проиллюстрируем интересующие рекурсивные шаблоны посредством конкретного типа данных — списка. Данные здесь определения в действительности являются частными случаями данных в §4. Современные функциональные языки поддерживают следующее определение списка типа A:

$$A_* ::= Nil \mid Cons(A||A_*)$$

Рекурсивная структура этого определения используется при написании функций $\in A_* \to B$, которые уничтожают список; такие функции названы *катаморфизмами* (от греческого предлога $\kappa\alpha\tau\alpha$, означающего "вниз", как в слове "катастрофа"). *Анаморфизмы* — это функции $\in B \to A_*$ (от греческого предлога $\alpha\nu\alpha$, означающего "вверх", как в слове "анаболизм"), которые генерируют список типа A_* из источника B. Функции типа $A \to B$, чьё дерево вызовов имеет форму списка, названы *хиломорфизмами* (в аристотелевской философии о том, что форма и материя суть одно, $\upsilon\lambda o\sigma$ означает "пыль" или "материя").

Катаморфизмы

Пусть $b \in B$ и $\oplus \in A||B \to B$, тогда списочный катаморфизм $h \in A_* \to B$ есть функция следующего вида:

$$\begin{array}{rcl} h\ Nil & = & b \\ h\ (Cons(a, as)) & = & a \oplus (h\ as) \end{array} \tag{1}$$

В нотации Бёрда и Уодлера [5] мы могли бы написать $h = foldr\ b\ (\oplus)$. Мы записываем катаморфизмы, размещая соответствующие составляющие между так называемых банановых скобок:

$$h = (\!|b, \oplus|\!) \tag{2}$$

В бесчисленных функциях для работы со списками легко опознаются катаморфизмы, например, $length \in A_* \to Num$ или $filter\ p \in A_* \to A_*$, где $p \in A \to bool$.

$$length = (\!|0, \oplus|\!),\ \text{где}\ a \oplus n = 1 + n$$

$$\begin{array}{rcll} filter\ p & = & (\!|Nil, \oplus|\!), & \\ & & \text{где}\ a \oplus as & = & Cons(a, as), & p\ a \\ & & & = & as, & \neg p\ a \end{array}$$

Отделение шаблона рекурсии для катаморфизма $(\!|\_|\!)$ от его компонентов b и $\oplus$ позволяет рассуждать о катаморфных программах в алгебраической манере. Например, *Закон Слияния* для списочных катаморфизмов читается так:

$$f \circ (\!|b, \oplus|\!) = (\!|c, \oplus|\!) \Leftarrow f\ b = c \wedge f(a \oplus as) = a \otimes (f\ as)$$

Без специальной нотации для катаморфизмов, такой как $(\!|\_|\!)$ или $foldr$, нам бы пришлось формулировать закон слияния следующим образом.

Пусть h, g определены выражением

$$\begin{array}{rcl} h\ Nil & = & b \\ h\ (Cons(a, as)) & = & a \oplus (h\ as) \end{array} \qquad \begin{array}{rcl} g\ Nil & = & c \\ g\ (Cons(a, as)) & = & a \otimes (g\ as) \end{array}$$

тогда $f \circ h = g$ если $f\ b = c$ и $f\ (a \oplus as) = a \otimes (f\ as)$

Какой неуклюжий способ записи такого простого алгебраического свойства.

Анаморфизмы

Если предикат $p \in B \to bool$ и функция $g \in B \to A||B$, то списочный анаморфизм $h \in B \to A_*$ определен как:

$$\begin{array}{rcll} h\ b & = & Nil, & p\ b \\ & = & Cons(a, h\ b'), & \text{иначе} \\ & & \text{где}\ (a, b') = g\ b & \end{array} \tag{3}$$

Анаморфизмы не так широко известны в фольклоре функционального программирования, они названы *развертками (unfold)* Бёрдом и Уодлером, которые потратили буквально несколько слов на них. Мы обозначаем анаморфизмы заключением соответствующих компонентов между вогнутых линз:

$$h = [\!(g, p)\!] \tag{4}$$

Многие важные функции над списками — анаморфизмы; например, функция $zip \in A_*||B_* \to (A||B)_*$, которая "застегивает" пару списков в список пар.

$$\begin{aligned} zip &= [\![g, p]\!] \\ p\,(as, bs) &= (as = Nil) \vee (bs = Nil) \\ g\,(Cons(a, as), Cons(b, bs)) &= ((a, b), (as, bs)) \end{aligned}$$

Другой анаморфизм — функция $iterate\,f$, которая для данного a конструирует бесконечный список повторяющихся применений f к a.

$$iterate\,f = [\![g, false^\bullet]\!], \text{где } g\,a = (a, f\,a)$$

Мы используем $c^\bullet$ для обозначения константной функции $\lambda x.c$.

Если $f \in A \to B$, функция–отображение $f_* \in A_* \to B_*$ применяет f к каждому элементу данного списка.

$$\begin{aligned} f_* Nil &= Nil \\ f_*(Cons(a, as)) &= Cons(f\,a, f_* as) \end{aligned}$$

Поскольку список присутствует в обеих частях её типа, мы можем подозревать, что функция–отображение может быть записана и как катаморфизм, и как анаморфизм. И в самом деле это так. Как катаморфизм: $f_* = (\!|Nil, \oplus|\!)$, где $a \oplus bs = Cons(f\,a, bs)$, и как анаморфизм $f_* = [\![g, p]\!]$, где $p\,as = (as = Nil)$ и $g\,(Cons(a, as)) = (f\,a, as)$.

Хиломорфизмы

Рекурсивная функция $h \in A \to C$, дерево вызовов которой изоморфно списку, т.е. линейно-рекурсивная функция, называется хиломорфизмом. Пусть $c \in C$, $\oplus \in B||C \to C$, $g \in A \to B||A$ и $p \in A \to bool$, тогда следующие строки определяют хиломорфизм h

$$\begin{aligned} h\,a &= c, && p\,a \\ &= b \oplus (h\,a'), && \text{иначе} \\ &\quad \text{где } (b, a') = g\,a \end{aligned} \tag{5}$$

Это точно такая же структура, как и анаморфизм, за исключением того, что Nil заменен на , а $Cons$ на $\oplus$. Мы записываем хиломорфизмы, заключая соответствующие части в конверты.

$$h = [\![(c, \oplus), (g, p)]\!] \tag{6}$$

Хиломорфизм соответствует композиции анаморфизма, явно строящего дерево вызовов как структуру данных, и катаморфизма, сворачивающего этот объект данных в требуемое значение.

$$[\![(c, \oplus), (g, p)]\!] = (\!|c, \oplus|\!) \circ [\![g, p]\!]$$

Доказательство этого равенства будет дано в §15[1].

Архетипичным хиломорфизмом является функция факториала:

$$\begin{aligned} fac &= [\![(1, \times), (g, p)]\!] \\ p\,n &= n = 0 \\ g\,(1 + n) &= (1 + n, n) \end{aligned}$$

[1]Имеется в виду §5. [Прим перев.]

Параморфизмы

Хиломорфное определение факториала, возможно, корректно, но неудовлетворительно с теоретический точки зрения, поскольку не оно является индуктивно определенным на типе данных $num ::= 0 \mid 1 + num$. Однако, не существует некоего "простого" φ, такого, что $fac = (\!|\varphi|\!)$. Проблема с факториалом в том, что он "съедает свой аргумент и в то же время сохраняет его" [27]; прямолинейное катаморфное решение могло бы работать с функцией fac', возвращающей пару $(n, n!)$, чтобы вычислить $(n+1)!$.

Параморфизмы были исследованы Меертенсом [19] для охвата этого шаблона простейшей рекурсии. Для типа num параморфизм есть функция h вида:

$$\begin{array}{rcl} h\,0 & = & b \\ h\,(1+n) & = & n \oplus (h\,n) \end{array} \tag{7}$$

Для списков параморфизм есть функция h вида:

$$\begin{array}{rcl} h\,Nil & = & b \\ h\,(Cons(a, as)) & = & a \oplus (as, h\,as) \end{array}$$

Мы записываем параморфизмы, заключая их соответствующие компоненты в колючую проволку $h = (\!|b, \oplus|\!)$, таким образом, мы можем записать $fac = (\!|1, \oplus|\!)$, где $n \oplus m = (1+n) \times m$. Функция $tails \in A_* \to A_{**}$, которая возвращает список всех хвостовых сегментов данного списка, определена как параморфизм $tails = (\!|Cons(Nil, Nil), \oplus|\!)$, где $a \oplus (as, tls) = Cons(Cons(a, as), tls)$.

3 Алгебраические типы данных

В предыдущем разделе мы задали нотацию описания некоторых рекурсивных шаблонов, связанную с единственным типом — списком. Чтобы определить понятия ката-, ана-, хило- и параморфизма для произвольных типов данных, мы теперь представляем общую теорию типов данных и функций над ними. Для этого мы описываем рекурсивный тип данных (также называемый "алгебраическим" типом данных в языке Miranda) для использования его в качестве наименьшей неподвижной точки функтора[2].

Функторы

Бифунктор $\dagger$ есть бинарная операция, преобразующая типы в типы и функции в функции так, что если $f \in A \to B$ и $g \in C \to D$, то $f \dagger g \in A \dagger C \to B \dagger D$, и сохраняющая свойства идентичности и композиции:

$$\begin{array}{rcl} id \dagger id & = & id \\ f \dagger g \circ h \dagger j & = & (f \circ h) \dagger (g \circ j) \end{array}$$

Бифункторы обозначаются символами $\dagger$,$\ddagger$,§,...

Монофунктор есть унарная операция над типами ${}_F$, которая также является операцией и над функциями, ${}_F \in (A \to B) \to (A_F \to B_F)$, сохраняющая свойства идентичности и

[2]Мы даем определения различных понятий теории категорий только для частного случая — категории CPO. Так, "функторы" на самом деле эндофункторы, и так далее.

композиции. Мы используем $_F$,$_G$,... для обозначения монофункторов. В силу обозначения A_* мы пишем применение функтора в постфиксной форме: A_F. В §5 мы покажем, что $_*$ действительно является функтором.

Типы данных всех настоящих функциональных языков могут быть определены при помощи следующих базовых функторов.

Произведение Произведение (ленивое) $D||D'$ двух типов D и D' и его операция $||$ над функциями определяются так:

$$\begin{array}{rcl} D||D' & = & (d, d' \,|\, d \in D, d' \in D') \\ (f||g)\,(x, x') & = & (f\,x, g\,x') \end{array}$$

С функтором $||$ тесно связаны комбинаторы проекции и заключения в кортеж:

$$\begin{array}{rcl} \grave{\pi}\,(x, y) & = & x \\ \acute{\pi}\,(x, y) & = & y \\ (f \triangledown g)\,x & = & (f\,x, g\,x) \end{array}$$

При помощи $\grave{\pi}$, $\acute{\pi}$ и $\triangledown$ мы можем выразить $f||g$ как $f||g = (f \circ \grave{\pi}) \triangledown (g \circ \acute{\pi})$. Так же мы можем определить $\triangledown$, используя $||$ и дублирующий комбинатор $\bigtriangleup x = (x, x)$, поскольку $f \triangledown g = f||g \circ \bigtriangleup$.

Сумма Сумма $D|D'$ типов D и D' и операция $|$ над функциями определены так:

$$\begin{array}{rcl} D|D' & = & (0||D) \cup (1||D') \cup \bot \\ (f|g)\ \bot & = & \bot \\ (f|g)\,(0, x) & = & (0, f\,x) \\ (f|g)\,(1, x') & = & (1, g\,x') \end{array}$$

Произвольно выбранные числа 0 и 1 используются в качестве "тэгов" значений двух слагаемых так, чтобы их можно было отличить друг от друга. Тесно связаны с функтором $|$ комбинаторы инъекции и выбора:

$$\begin{array}{rcl} \grave{\imath}\,x & = & (0, x) \\ \acute{\imath}\,y & = & (q, y) \\ f \vartriangle g\ \bot & = & \bot \\ f \vartriangle g\,(0, x) & = & f\,x \\ f \vartriangle g\,(1, y) & = & g\,y \end{array}$$

при помощи которых мы можем записать $f|g = (\grave{\imath} \circ f) \vartriangle (\acute{\imath} \circ g)$. Используя $\bigtriangledown$, который удаляет тэги из своего аргумента, $\bigtriangledown\ \bot{=}\bot$ и $\bigtriangledown\,(i, x) = x$, мы можем определить $f \vartriangle g = \bigtriangledown \circ f|g$.

Стрелка Операция $\to$, формирующая пространство $D \to D'$ непрерывных функций из D в D', обладает операцией над функциями — "оборачивающей" функцией:

$$(f \to g)\,h = g \circ h \circ f$$

Часто мы будем использовать альтернативную запись $(g \leftarrow f)\,h = g \circ h \circ f$, где мы уже заменили стрелку так, что не требуется перемещать аргументы, таким образом локализуя

изменения, полученные в результате вычислений. Функционал $(f \overset{F}{\leftarrow} g)\, h = f \circ h_F \circ g$ помещает его аргумент $_F$ между f и g.

С $\to$ тесно сязаны комбинаторы:

$$\begin{aligned} curry\, f\, x\, y &= f\,(x,y) \\ uncurry\, f\,(x,y) &= f\, x\, y \\ eval\,(f,x) &= f\, x \end{aligned}$$

Заметьте, что $\to$ — это контравариантный функтор относительно первого аргумента, т.е. $(f \to g) \circ (h \to j) = (h \circ f) \to (g \circ j)$.

Идентичность, константы Функтор идентичности $\mathbf{I}$ определен на типах как $D_{\mathbf{I}} = D$, и на функциях как $f_{\mathbf{I}} = f$. Для любого типа D существует функтор с таким же именем $\underline{D}$, действие которого над объектами задается как равенством $C\underline{D} = D$, а над функциями - $f\underline{D} = id$.

Поднятие Для монофункторов $_F$, $_G$ и бифунктора $\dagger$ мы определяем монофункторы $_{FG}$ и $_F\dagger_G$

$$\begin{aligned} x(_{FG}) &= (x_F)_G \\ x(_F\dagger_G) &= (x_F) \dagger (x_G), \end{aligned}$$

где x может быть как типом, так и функцией.

В силу первого уравнения нам не требуется записывать скобки в x_{FG}. Обратите внимание, что в выражении $(_F\dagger_G)$ бифунктор $\dagger$ "поднят" для применения его к функторам, а не к объектам; $_f\dagger_G$ сам по себе является монофунктором.

Частичное применение Аалогично частичному применению бинарных операций $(a\oplus)\, b = a \oplus b$ и $(\oplus b)\, a = a \oplus b$ определим частичное применение бифункторов $\dagger$:

$$\begin{aligned} (A\dagger) &= \underline{A}\dagger_{\mathbf{I}} \\ (f\dagger) &= f \dagger id, \end{aligned}$$

откуда $B(A\dagger) = A \dagger B$ и $f(A\dagger) = id \dagger f$. Похожим образом мы можем определить частичное применение $\dagger$ по его второму аргументу, т.е. $(\dagger B)$ и $(\dagger f)$.

Несложно проверить следующие два свойства частично примененных функторов:

$$(f\dagger) \circ g(A\dagger) = g(B\dagger) \circ (f\dagger) \quad \text{для всех } f \in A \to B \tag{8}$$

$$(f\dagger) \circ (g\dagger) = ((f \circ g)\dagger) \tag{9}$$

Приняв $f \dagger g = g \to f$, так что $(f\dagger) = (f \circ)$, получим несколько изящных законов для композиции функций.

Законы базовых комбинаторов

Существуют самые различные уравнения, включающие вышеуказанные комбинаторы, мы лишь констатируем некоторые из них. В выражениях композиция функций имеет низший

приоритет, а || - больший, чем |.

$$\begin{array}{rcl@{\qquad}rcl}
\grave{\pi} \circ f||g & = & f \circ \grave{\pi} & f|g \circ \grave{i} & = & \grave{i} \circ f \\
\grave{\pi} \circ f \triangledown g & = & f & f \vartriangle g \circ \grave{i} & = & f \\
\acute{\pi} \circ f||g & = & g \circ \acute{\pi} & f|g \circ \acute{i} & = & \acute{i} \circ g \\
\acute{\pi} \circ f \triangledown g & = & g & f \vartriangle g \circ \acute{i} & = & g \\
(\grave{\pi} \circ h) \triangledown (\acute{\pi} \circ h) & = & h & (h \circ \grave{i}) \vartriangle (h \circ \acute{i}) & = & h \Leftarrow h \text{ строгая} \\
\grave{\pi} \triangledown \acute{\pi} & = & id & \grave{i} \vartriangle \acute{i} & = & id \\
f||g \circ h \triangledown j & = & (f \circ h) \triangledown (g \circ j) & f \vartriangle g \circ h|j & = & (f \circ h) \vartriangle (g \circ j) \\
f \triangledown g \circ h & = & (f \circ h) \triangledown (g \circ h) & f \circ g \vartriangle h & = & (f \circ g) \vartriangle (f \circ h) \Leftarrow f \text{строгая} \\
f||g = h||j & \equiv & f = h \wedge g = j & f|g = h|j & \equiv & f = h \wedge g = j \\
f \triangledown g = h \triangledown j & \equiv & f = h \wedge g = j & f \vartriangle g = h \vartriangle j & \equiv & f = h \wedge g = j
\end{array}$$

Элегантно выглядит связывающий $\triangledown$ и $\vartriangle$ *закон соответствия*:

$$(f \triangledown g) \vartriangle (h \triangledown j) = (f \vartriangle h) \triangledown (g \vartriangle j) \tag{10}$$

Разное

Тип, содержащий единственный элемент, обозначается как $\mathbf{1}$ и может быть использован для моделирования констант типа A функциями без аргументов типа $\mathbf{1} \to A$. Единственный член типа $\mathbf{1}$ называется *пустым* и обозначается $()$.

В некоторых примерах мы используем для данного предиката $p \in A \to bool$ функцию

$$\begin{array}{rcll}
p? & \in & A \to A|A & \\
p?\, a & = & \bot, & p\, a = \bot \\
 & = & \grave{i}\, a, & p\, a = true \\
 & = & \acute{i}\, a, & p\, a = false
\end{array}$$

таким образом, $f \vartriangle g \circ p?$ моделирует известное условное выражение $\mathbf{if}\, p\, \mathbf{then}\, f\, \mathbf{else}\, g\, \mathbf{fi}$. Функция $VOID$ отображает свой аргумент на пустое значение: $VOID\, x = ()$. Вот некоторые закономерности, выполняющиеся для таких функций:

$$\begin{array}{rcl}
VOID \circ f & = & VOID \\
p? \circ x & = & x|x \circ (p \circ x)?
\end{array}$$

Чтобы сделать рекурсию явной, мы используем оператор $\mu \in (A \to A) \to A$, определенный следующим образом:

$$\mu\, f = x, \text{ где } x = f\, x$$

Предполагается, что рекурсия (вроде $x = f\, x$) определена в метаязыке.

Пусть $_F$, $_G$ - функторы, и $\varphi_A \in A_F \to A_G$ для любого типа A. Такие функции, как φ, называются *полиморфными*. *Естественное преобразование* - это семейство функций φ_A (опуская подстрочные символы везде, где возможно) таких, что:

$$\forall f : f \in A \to B : \varphi_B \circ f_F = f_G \circ \varphi_A \tag{11}$$

В качестве удобного сокращения для (11) мы используем $\varphi \in_F \!\!\to_G$ для обозначения того, что φ является естественным преобразованием. Теорема Уодлера, де Брейна (deBruin) и Рейнольдса (Reynolds) [28, 9, 22] из "Бесплатных теорем" утверждает, что любая функция, определимая в полиморфном λ-исчислении, есть естественное преобразование. Если φ определить через μ, то можно сделать вывод, что (11) справедливо для строгих f.

Рекурсивные типы

После всех этих рассуждений о функторах мы наконец достаточно вооружились, чтобы абстрагироваться от особенностей списков и формализовать рекурсивно определенные типы данных в целом.

Пусть $_F$ — монофунктор, непрерывный на функциях, т.е. любой монофунктор, определенный с использованием описанных выше базовых функторов или любых функторов-отображений, описанных в §5. Тогда существуют тип L и две строгие функции $in_F \in L_F \to L$ и $out_F \in L \to L_F$ (опуская подстрочные символы везде, где возможно), являющиеся обратными и дополняющими друг друга: $id = \mu(in \overset{F}{\leftarrow} out)$ [6, 23, 16, 24, 30, 12]. Мы обозначем пару (L, in) как μ_F и называем её "наименьшей неподвижной точкой $_F$". Поскольку in и out обратны друг другу, получаем, что L_F изоморфен L, и действительно, L — в силу изоморфизма — неподвижная точка $_F$.

Например, приняв $X_L = \mathbf{1}|A||X$, получаем, что $(A_*, in) = \mu_L$ определяет тип данных — список элементов A для любого типа A. Положив $Nil = in \circ \grave{\imath} \in \mathbf{1} \to A_*$ и $Cons = in \circ \acute{\imath} \in A||A_* \to A_*$, получим более знакомое $(A_*, Nil \vartriangle Cons) = \mu_L$. Другой пример типа данных, двоичные деревья с листьями типа A, получается, если взять наименьшую неподвижную точку $X_T = \mathbf{1}|A|X||X$. Обратные списки с элементами типа A, или snoc-списки, как их еще иногда называют, есть наименьшая неподвижная точка $XL = \mathbf{1}|X||A$. Натуральные числа определяются как наименьшая неподвижная точка $X_N = \mathbf{1}|X$.

4 Схемы рекурсии

Теперь, поскольку мы задали общий способ определения рекурсивных типов данных, мы можем определить ката-, ана-, хило- и парамофизмы для произвольных типов данных. Пусть $(L, in) = \mu_F$, $\varphi \in A_F \to A$, $\psi \in A \to A_F$, $\xi \in (A||L)_F \to A$, тогда:

$$(\!|\varphi|\!)_F = \mu(\varphi \overset{F}{\leftarrow} out) \tag{12}$$

$$[\!(\psi)\!]_F = \mu(in \overset{F}{\leftarrow} \psi) \tag{13}$$

$$[\![\varphi, \psi]\!]_F = \mu(\varphi \overset{F}{\leftarrow} \psi) \tag{14}$$

$$\langle\!|\xi|\!\rangle_F = \mu(\lambda f.\xi \circ (id \triangledown f)_F \circ out) \tag{15}$$

Когда не возникает двусмысленности, мы опускаем подстрочные символы.

Определение (13) соответствует определению, данному в §2; раньше мы писали $(\!|e, \oplus|\!)$, теперь — $(\!|e^\bullet \vartriangle (\oplus)|\!)$.

Определение (14) соответствует ранее данному неформальному; запись $[\![g, p]\!]$ из §2 теперь становится $[\![(VOID|g) \circ p?]\!]$.

Определение (15) соответствует ранее данному; принимая $\varphi = c^\bullet \vartriangle \oplus$ и $\psi = (VOID|g) \circ p?$, превращаем $[\![(c^\bullet, \oplus), (g, p)]\!]$ в $[\![\varphi, \psi]\!]$.

Определение (15) соответствует описанию параморфизмов, как оно дано в §2 в том смысле, что $\langle\!|b, \oplus|\!\rangle$ равно $\langle\!|b^\bullet \vartriangle (\oplus)|\!\rangle$.

Законы вычисления программ

Предоставляя набор законов для определенных выше жестких шаблонов рекурсии, мы поощряем их использование, а не использование явной рекурсии программистом. Для каждого Ω-морфизма, где $\Omega \in \{$ката, ана, пара$\}$, мы задаем *правило вычисления*, которое показы-

вает, как такой морфизм может быть вычислен, *Уникальное Свойство* (УС) — индуктивное доказательство того, что данная функция является Ω-морфизмом, и *закон слияния*, который показывает, когда композиция функции и Ω-морфизма снова является Ω-морфизмом. Все эти законы могут быть доказаны простым выводом с использованием следующих свойств рекурсивных функций. Первое свойство — "бесплатная теорема" для оператора неподвижной точки $\mu \in (A \to A) \to A$

$$f\ (\mu g) = \mu h \ \Leftarrow\ f \text{ строгая} \wedge f \circ g = h \circ f \tag{16}$$

Теорема (16) появляется под разными названиями во многих источниках[3] [20, 8, 2, 15, 7, 25, 13, 31]. В этой статье она будет называться *слиянием неподвижной точки.*

От требования строгости в (16) иногда можно избавиться:

$$f\ (\mu g) = f'\ (\mu g') \ \Leftarrow\ f\ \bot = f'\ \bot\ \wedge f \circ g = h \circ f \wedge f' \circ g' = h \circ f' \tag{17}$$

Запись неподвижной точки предиката $P(g, g') \equiv f\ = f'\ g'$ будет доказательством (17).

Мы доказали, что хиломорфизм может быть разделен на ана- и катаморфизм, и показали, каким образом вычисление может быть преобазовано внутри хиломорфизма. Некторые следствия показывают зависимость между ката- и анаморфизмами. Эти законы не являются верными в категории SET. Законы для хиломорфизмов вытекают из следующей теоремы:

$$\mu(f \overset{F}{\leftarrow} g) \circ \mu(h \overset{F}{\leftarrow} j) \ \Leftarrow\ g \circ h = id \tag{18}$$

Катаморфизмы

Правило вычисления *Правило вычисления* для катаморфизмов следует из свойства неподвижной точки $x = \mu f \Rightarrow x = f\ x$:

$$(\!|\varphi|\!) \circ in = \varphi \circ (\!|\varphi|\!)_L \tag{CataEval}$$

Оно показывает, как вычислить применение $(\!|\varphi|\!)$ к произвольному элементу L (который возвращает конструктор in), а именно: применить $(\!|\varphi|\!)$ рекурсивно к аргументу in, а потом φ к результату.

Для списков $(A_*, Nil \triangle Cons) = \mu_L$, где $X_L = \mathbf{1}|A||X$ и $f_L = id|id||f$ с катаморфизмом $(c \triangle \oplus)$ правило вычисления читается как

$$(\!|c \triangle \oplus|\!) \circ Nil \ = \ c \tag{19}$$

$$(\!|c \triangle \oplus|\!) \circ Cons \ = \ \oplus \circ id||(\!|c \triangle \oplus|\!) \tag{20}$$

т.е. представляет собой свободную от переменных формулировку (1). Заметьте, что конструкторы (здесь $Nil \triangle Cons$) используются для сопоставления параметров с образцом.

УС катаморфизмов *Уникальное свойство* может быть использовано для доказхательства идентичности двух функций без явного применения индукции.

$$f = (\!|\varphi|\!) \equiv f \circ \bot = (\!|\varphi|\!) \circ \bot \wedge f \circ in = \varphi \circ f_L \tag{CataUP}$$

[3] Другие ссылки также приветствуются.

Типичное индуктивное доказательство, показывающее $f = (\!|\varphi|\!)$ предполагает слудующие шаги. Проверим базу индукции: $f \circ \bot = (\!|\varphi|\!) \circ \bot$. Внесем индуктивную гипотезу $f_L = (\!|\varphi|\!)_L$, затем рассуждая

$$\begin{array}{ll} & f \circ in = \ldots = \varphi \circ f_L \\ = & \quad \text{гипотеза индукции} \\ & \varphi \circ (\!|\varphi|\!)_L \\ = & \quad \text{правило вычисления}(CataEval) \\ & (\!|\varphi|\!) \circ in \end{array}$$

заключаем, что $f = (\!|\varphi|\!)$. Схема такого доказательства раз и навсегда получена и включена в закон (CataUP). Таким образом мы спаслись от стандартных ритуальных шагов; остаются последние два шага из предыдущего расчета плюс утверждение, что "индуктивное" доказательство готово.

Правая часть доказательства (CataUP) прямо следует из правила вычисления катаморфизмов. Для левой части мы используем теорему слияния неподвижной точки (17) с $f := (f \circ)$, $g := g' := in \overset{L}{\leftarrow} out$ и $f' := (\!|\varphi|\!)$. Это дает нам $f \circ \mu(in \overset{L}{\leftarrow} out) = (\!|\varphi|\!) \circ \mu(in \overset{L}{\leftarrow} out)$, и, поскольку $\mu(in \overset{L}{\leftarrow} out) = id$, то все готово.

Закон слияния для катаморфизмов *Закон слияния* для катаморфизмов можно использваоть для преобразования композиции функции и катаморфизма в единий катаморфизм так, чтобы не использовать промежуточные значения. Иногда закон используется чтобы, наоборот, разделить функцию на части, подготавливая последующие оптимизации.

$$f \circ (\!|\varphi|\!) = (\!|\psi|\!) \Leftarrow f \circ \bot = (\!|\psi|\!) \circ \bot \wedge f \circ \varphi = \psi \circ f_L \qquad \text{(CataFusion)}$$

Закон слияния может быть доказан с использванием теоремы слияния неподвижной точки (17) с $f := (f \circ)$, $g := \varphi \overset{L}{\leftarrow} out$, $g' := in \overset{L}{\leftarrow} out$ и $f' := ((\!|\psi|\!) \circ)$.

Небольшая вариация закона слияния получается заменой условия $f \circ \bot = (\!|\psi|\!) \circ \bot$ на $f \circ \bot = \bot$, т.е. f - строгая функция.

$$f \circ (\!|\varphi|\!) = (\!|\psi|\!) \Leftarrow f \text{ строгая} \wedge f \circ \varphi = \psi \circ f_L \qquad \text{(CataFusion')}$$

Этот закон следует из (16). В практических расчетах этот последний закон более значим, поскольку его условия применения в целом легче проверить.

Инъективные функции есть катаморфизмы Пусть $f \in A \to B$ — строгая функция с левой обратной функцией g, тогда для любого $\varphi \in A_F \to A$ имеем

$$f \circ (\!|\varphi|\!) = (\!|f \circ \varphi \circ g_F|\!) \Leftarrow f \text{ строгая} \wedge g \circ f = id \qquad (21)$$

Принимая $\varphi = in$, сразу же получаем, что любая строгая инъективная функция может быть записана как катаморфизм.

$$f = (\!|f \circ in \circ g_f|\!)_F \Leftarrow f \text{ строгая} \wedge g \circ f = id \qquad (22)$$

Используя последний результат, мым можем записать out через in, поскольку $out = (\!|out \circ in \circ in_L|\!) = (\!|in_L|\!)$.

Катаморфизм сохраняет свойство строгости Все данные законы для катаморфизмов демонстрируют важность свойства строгости, или поведения функции относительно $\perp$. По этой причине следующий "анализатор строгости для бедняков" может быть часто и удачно применен.

$$\mu F \circ \perp = \perp \Leftarrow \forall f :: F\ f \circ \perp = \perp \tag{23}$$

(23) доказывается индукцией неподвижной точки над $P(F) \equiv F \circ \perp = \perp$.

Конкретно для катаморфизмов имеем

$$(\!|\varphi|\!)_L \circ \perp = \perp \equiv \varphi \circ \perp = \perp$$

если $_L$ сохраняет свойство строгости. Левая часть доказательства прямо следует из (23) и определения катаморфизма. Обратное доказательство выглядит так:

$$\begin{array}{ll} & \perp \\ = & \quad \text{предпосылка} \\ & (\!|\varphi|\!) \circ \perp \\ = & \quad in \circ \perp = \perp \\ & (\!|\varphi|\!) \circ in \circ \perp \\ = & \quad \text{правило вычисления} \\ & \varphi \circ (\!|\varphi|\!)_L \circ \perp \\ = & \quad {}_L \text{ сохраняет свойство строгости} \\ & \varphi \circ \perp \end{array}$$

Примеры

Свертка-развертка Многие преобразования, обычно выполняемые при помощи техники развертка-упрощение-свертка, могут быть пересмотрены с использованием слияния. Пусть $(Num_*, Nil \triangle Cons) = \mu_L$, где $X_L = \mathbf{1}|Num||X$ и $f_L = id|id||f$, — тип списка натуральных чисел. Используя слияние, мы получаем эффективную версию функции $sum \circ squares$, где $sum = (\!|0^\bullet \triangle +|\!)$ и $squares = (\!|Nil \triangle (Cons \circ SQ||id)|\!)$. Поскольку sum — строгая функция, мы просто начинаем рассуждения, имея целью нахождение ψ, удовдетворяющего условиям (CataFusion').

$$\begin{array}{ll} & sum \circ Nil \triangle (Cons \circ S||id)^4 \\ = & \\ & (sum \circ Nil) \triangle (sum \circ Cons \circ SQ||id) \\ = & \\ & Nil \triangle ((+) \circ id||sum \circ SQ||id) \\ = & \\ & Nil \triangle ((+) \circ SQ||id \circ id||sum) \\ = & \\ & Nil \triangle ((+) \circ SQ||id) \circ sum_L \end{array}$$

И делаем вывод, что $sum \circ squares = (\!|Nil \triangle ((+) \circ SQ||id)|\!)$.

Чуть более сложная задача — получить простое решение для

$$average = DIV \circ sum \triangledown length$$

[4]Вероятно, имелось в виду SQ вместо S. [Прим. перев.]

Используя лемму объединения в кортеж Фоккинги [10], имеем

$$(\!|\varphi|\!)_L \triangledown (\!|\psi|\!) = (\!|(\varphi \circ \grave{\pi}_L) \triangledown (\psi \circ \acute{\pi}_L)|\!)$$

Простой расчет показывает, что $average = DIV \circ (\!|(0^\bullet \vartriangle (+) \circ id || \grave{\pi}) \triangledown (0^\bullet \vartriangle (+1) \circ \acute{\pi})$

Накопление аргументов Одним из важных хитростей программистов-функциональщиков является техника *накопления аргументов*, когда к функции добавляется дополнительный параметр для хранения результата вычисления. Хотя здесь эта техника определена в терминах катаморфизмов на списках, точно так же она может быть применена к другим типам данных и другим морфизмам.

$$(\!|c^\bullet \vartriangle \oplus|\!)\ l = (\!|(c\otimes)^\bullet \vartriangle \ominus|\!)\ l\ \nu_\oplus, \text{ где } (a \ominus f)\ b = f\ (a \odot b) \tag{24}$$
$$\Leftarrow$$
$$a \otimes \nu_\oplus = a \wedge \ \bot \otimes a = \bot \wedge (a \oplus b) \otimes c = b \otimes (a \odot c)$$

Теорема (24) следует из закона слияния, если принять $Accu \circ (\!|c^\bullet \vartriangle \oplus|\!) = (\!|(c\oplus)^\bullet \vartriangle \ominus|\!)$ и $Accuab = a \otimes b$.

Имея наивное квадратичное определение $reverse \in A_* \to A_*$ как катаморфизма $(\!|Nil^\bullet \vartriangle \oplus|\!)$, где $a \oplus as = as +\!\!+ (Cons(a, Nil))$, мы можем вывести алгоритм, выполняющийся за линейное время, подставляя в (24) $\oplus := +\!\!+$ и $\odot := Cons$, чтобы получить функцию, которая собирает обращенный список как дополнительный аргумент: $(\!|id \vartriangle \ominus|\!)$, где $(a \ominus as)\ bs = as\ (Cons(a, bs))$. Здесь $+\!\!+$ обозначет функцию, соединяющую два списка, определенну как $as +\!\!+ bs = (\!|id^\bullet \vartriangle \oplus|\!)\ as\ bs$, где $a \oplus f\ bs = Cons(a, f\ bs)$.

Вообще катаморфизмы более высокого типа $L \to (I \to S)$ формируют интересный класс сами по себе, так как они соответствуют *атрибутным грамматикам* [11].

Анаморфизмы

Правило вычисления Правило вычисления анаморфизмов задается выражением

$$out \circ [\![\psi]\!] = [\![\psi]\!]_L \circ \psi \tag{AnaEval}$$

Здесь говорится, что результат произвольного применения $[\![\psi]\!]$ выглядит так: составляющие, полученные применением out, могут быть так же получены применением сначала ψ, а затем рекурсивным применением $[\![\psi]\!]_L$ к результату.

Анаморфизмы и так уже всем надоели, чтоб объянять их еще раз. Проиллюстрируем (AnaEval) для списков:

$$\begin{aligned} hd &= \bot \vartriangle \grave{\pi} \circ out \\ tl &= \bot \vartriangle \acute{\pi} \circ out \\ is\_nil &= true^\bullet \vartriangle false^\bullet \circ out \end{aligned}$$

Принимая, что $f = [\![VOID | (h \triangledown t) \circ p?]\!]$, после недолгих вычислений находим, что

$$\begin{aligned} is\_nil \circ f &= p \\ hd \circ f &= h \Leftarrow \neg p \\ tl \circ f &= t \Leftarrow \neg p \end{aligned}$$

что соответствует описанию $unfold$, данному Бёрдом и Уодлером [5] на странице 173.

УС анаморфизмов Уникальное свойство анаморфизмов немного проще, чем в случае катаморфизмов, поскольку не нужно проверять базовый случай.

$$f = [\![\varphi]\!] \equiv out \circ f = f_L \circ \varphi \tag{AnaUP}$$

Чтобы доказать его, мы можем использовать теорему слияния неподвижной точки (16), подставляя $f := (\circ f)$, $g := in \overset{L}{\leftarrow} out$ и $h := in \overset{L}{\leftarrow} \psi$. Это дает нам $\mu(in \overset{L}{\leftarrow} out) \circ f = \mu(in \overset{L}{\leftarrow} \psi)$, и снова, покольку $\mu(in \overset{L}{\leftarrow} out) = id$, то доказательство готово.

Закон слияния для анаморфизмов Требование строгости, нужное для катаморфизмов, может быть отброшено в случае анаморфизмов. Дуальное к условию строгости $f \circ \ \bot = \bot$ условие $\bot \ \circ f = \bot$ всегда верно.

$$[\![\varphi]\!] \circ f = [\![\psi]\!] \ \Leftarrow \ \varphi \circ f = f_L \circ \psi \tag{AnaFusion}$$

Этот закон может быть доказан при помощи теоремы слияния неподвижной точки (16) с подстановкой $f := (\circ f)$, $g := in \overset{L}{\leftarrow} \varphi$ и $h := in \overset{L}{\leftarrow} \psi$.

Любая сюръективная функция есть анаморфизм Результаты (21) и (22) могут быть отображены на анаморфизмы. Пусть $f \in B \to A$ — сюръективная функция с левой обратной функцией g, тогда для любого $\psi \in A \to A_L$ имеем

$$[\![\psi]\!] \circ f = [\![g_L \circ \psi \circ f]\!] \ \Leftarrow \ f \circ g = id \tag{25}$$

поскольку $\psi \circ f = f_L \circ (g_L \circ \psi \circ f)$. Частный случай, когда ψ тождественно out, показывает, что любая сюръективная функция может быть записана как анаморфизм.

$$f = [\![g_L \circ out \circ f]\!]_L \ \Leftarrow \ f \circ g = id \tag{26}$$

Так как out — правая обратная функция in, мы можем выразить in через out: $in = [\![out_L \circ out \circ in]\!] = [\![out_L]\!]$.

Примеры

Если переформулировать её в "нотации линз", функция $iterate\ f$ станет такой:

$$iterate\ f = [\![\acute{i} \circ id \triangledown f]\!]$$

Имеем $[\![\acute{i} \circ id \triangledown f]\!] = [\![VOID | id \triangledown f \circ false^\bullet?]\!]$ ($= [\![id \triangledown f, false^\bullet]\!]$ в нотации из раздела 2).

Другая полезная функция над списками — $takewhile\ p$ — выбирает начальный сегмент списка наибольшей длины, все элементы которого удовлетворяют p. В обычной нотации:

$$\begin{array}{rcll} takewhile\ p\ Nil & = & Nil & \\ takewhile\ p\ (Cons\ a\ as) & = & Nil, & \neg p\ a \\ & = & Cons\ a\ (takewhile\ p\ as) & \text{иначе} \end{array}$$

Определение в виде анаморфизма может сначала выглядеть несколько обескураживающе:

$$takewhile\ p = [\![\grave{i} \vartriangle (VOID | id \circ (\neg p \circ \acute{\pi})?) \circ out]\!]$$

Функция $f\ while\ p$ повторяет применения f, пока предикат истинен:

$$f\ while\ p = takewhile\ p \circ iterate\ f$$

Используя закон слияния (и, скорее всего, после долгих расчетов), мы можем показать, что $f\ while\ p = [\![VOID | (id \triangledown f) \circ \neg p?]\!]$.

Хиломорфизмы

Разделение хиломорфизмов Чтобы доказать, что хиломорфизм может быть разделен на анаморфизм с последующим катаморфизмом

$$[\![\varphi, \psi]\!] = (\!|\varphi|\!) \circ [\!(\psi)\!] \qquad \text{(HyloSplit)}$$

мы можем использовать полную теорему слияния (18).

Закон перестановки Хиломорфизмы удобны тем, что возможность их разделения на ката- и анаморфизм позволяет нам использовать соответствующие законы слияния для внесения и вынесения вычислений из хиломорфизма. Записанные далее *закон перестановки* показывает, как вычисления могут быть переставлены внутри хиломорфизма.

$$[\![\varphi \circ \xi, \psi]\!]_L = [\![\varphi, \xi \circ \psi]\!]_M \Leftarrow \xi \in_L \to_M \qquad \text{(HyloSplit)}$$

Доказательство этой теоремы довольно прямолинейно.

$$\begin{array}{ll} & [\![\varphi \circ \xi, \psi]\!]_L \\ = & \quad \text{определение хиломорфизма} \\ & \mu(\lambda f.\varphi \circ \xi \circ f_L \circ \psi) \\ = & \quad \xi \in_L \to_M \\ & \mu(\lambda f.\varphi \circ f_M \circ \xi \circ \psi) \\ = & \quad \text{определение хиломорфизма} \\ & [\![\varphi, \xi \circ \psi]\!]_M \end{array}$$

Общеизвестно "мошеннический" пример тождества (HyloSplit) показывает, как линейные леворекурсивные функции могут быть преобразованы в праворекурсивные. Пусть $f_L = id|f||id$ и $f_R = id|id||f$ задают функторы, выражающие левую и правую рекурсии соттветственно, тогда, если $x \oplus y = y \oplus x$, имеем

$$\begin{array}{ll} & [\![c \vartriangle \oplus, f|(h \triangledown t) \circ p?]\!]_L \\ = & \\ & [\![c \vartriangle \ \oplus \ \circ SWAP, f|(h \triangledown t) \circ p?]\!]_L \\ = & \quad SWAP \in_L \to_R \\ & [\![c \vartriangle \oplus, SWAP \circ f|(h \triangledown t) \circ p?]\!]_R \\ = & \\ & [\![c \vartriangle \oplus, f|(t \triangledown h) \circ p?]\!]_R \end{array}$$

где $SWAP = id|(\acute{\pi} \triangledown \grave{\pi})$.

Связь ката- и анаморфизмов

Исходя из закона слияния и перестановки (HyloSplit), (HyloSplit) и того факта, что $(\!|\varphi|\!) = [\![\varphi, out]\!]$ и $[\!(\psi)\!] = [\![in, \psi]\!]$, мы можем вывести несколько интересных законов, связывающих ката- и анаморфизмы друг с другом.

$$(\!|in_M \circ \varphi|\!)_L = [\!(\varphi \circ out)\!]_M \Leftarrow {}_L \to_M \qquad (27)$$

Используя этот закон, легко показать, что

$$
\begin{aligned}
(\!|\varphi\psi|\!)_L &= (\!|\varphi|\!)_M \circ [\!(\psi \circ out_L)]\!]_M \Leftarrow \psi \in_L \to_M && (28)\\
&= (\!|\varphi|\!)_M \circ (\!|in_M \circ \psi|\!)_L \Leftarrow \psi \in_L \to_M && (29)\\
& && (30)\\
[\!(\varphi \circ \psi)]\!] &= (\!|in_M \circ \varphi|\!)_L \circ [\!(\psi)]\!]_L \Leftarrow \varphi \in_L \to_M && (31)\\
&= [\!(\varphi \circ out_L)]\!]_M \circ [\!(\psi)]\!]_L \Leftarrow \varphi \in_L \to_M && (32)
\end{aligned}
$$

Этот набор законов будет использован в §5.

Из полной теоремы слияния (18) можно вывести:

$$[\!(\psi)]\!]_L \circ (\!|\varphi|\!)_L = id \Leftarrow \psi \circ \varphi = id \tag{33}$$

Пример: Отражение двоичных деревьев

Тип двоичных деревьев с дистьями типа A задается выражением $(tree\ A, in) = \mu_L$, где $X_L = \mathbf{1}|A|X||X$ и $f_L = id|id|g||g$. Отражение двоичного дерева может быть задано как $reflect = (\!|in \circ SWAP|\!)$, где $SWAP = id|id|(\grave{\pi} \triangledown \grave{\pi})$. Простой расчет показывает, что $reflect \circ reflect = id$.

$$
\begin{array}{ll}
 & reflect \circ reflect\\
= & \quad SWAP \circ f_L = f_L \circ SWAP\\
 & [\!(SWAP \circ out)]\!] \circ (\!|in \circ SWAP|\!)\\
= & \quad SWAP \circ out \circ in \circ SWAP = id\\
 & id
\end{array}
$$

Параморфизмы

Правило вычисления для параморфизмов:

$$\langle\!|\varphi|\!\rangle \circ in = \varphi \circ (id \triangledown \langle\!|\varphi|\!\rangle)_L \tag{ParaEval}$$

УС параморфизмов похоже на УС катаморфизмов:

$$f = \langle\!|\varphi|\!\rangle \equiv f \circ \perp = \langle\!|\varphi|\!\rangle \circ \perp \wedge f \circ in = \varphi \circ (id \triangledown f)_L \tag{ParaUP}$$

Закон слияния для параморфизмов записывается как

$$f \circ \langle\!|\varphi|\!\rangle = \langle\!|\psi|\!\rangle \Leftarrow f \text{ строгая } \wedge f \circ \varphi = \psi \circ (id || f)_L \tag{ParaFusion}$$

Любая функция f (конечно, подходящего типа!) есть параморфизм.

$$f = \langle\!|f \circ in \circ \grave{\pi}_L|\!\rangle$$

Полезность этой теоремы видна из её доказательства.

$$
\begin{array}{ll}
 & \langle\!|f \circ in \circ \grave{\pi}_L|\!\rangle\\
= & \quad \text{определение (15)}\\
 & \mu(\lambda g.f \circ in \circ \grave{\pi}_L \circ (id \triangledown g)_L \circ out)\\
= & \quad \text{исчисление функторов}\\
 & \mu(\lambda g.f \circ in \circ out)\\
= & \\
 & f
\end{array}
$$

Пример: составление параморфизмов из ана- и катаморфизмов

Приятный вывод заключается в том, что любой параморфизм может быть записан как композиция ката- и анаморфизма. Пусть дано $(L, in) = \mu_L$, тогда

$$\begin{aligned} X_M &= (L||X)_L \\ h_M &= (id||h)_L \\ (M, IN) &= \mu_M \end{aligned}$$

Для натуральных чисел получаем $X_M = (Num||X)_L = \mathbf{1}|Num||X$, т.е. $(Num_*, in) = \mu_M$, что и представляет собой тип списка натуральных чисел.

Определим теперь $preds \in L \to M$ следующим образом:

$$preds = [\!(\Delta_L \circ out_L)\!]_M$$

Для натуральных чисел получаем $preds = [\!(id|\Delta \circ out)\!]$; если дано натуральное число $N = n$, то выражение $preds\, N$ даст список $[n-1, \ldots, 0]$.

Начнем рассуждение, используя $preds$:

$$\begin{aligned} & (\!|\varphi|\!)_M \circ preds \\ = & \\ & (\!|\varphi|\!)_M \circ [\!(\Delta_L \circ out_L)\!]_M \\ = & \\ & \mu(\lambda f.\varphi \circ f_M \circ \Delta_L \circ out_L) \\ = & \\ & \mu(\lambda f.\varphi \circ (id||f)_L \circ (id \triangledown id)_L \circ out_L) \\ = & \\ & \mu(\lambda f.\varphi \circ (id \triangledown f)_L \circ out_L) \\ = & \\ & \langle\!|\varphi|\!\rangle_L \end{aligned}$$

Таким образом, $\langle\!|\varphi|\!\rangle_L = (\!|\varphi|\!)_M \circ preds$. Поскольку $(\!|IN|\!)_M = id$, мы сразу получаем $preds = \langle\!|IN|\!\rangle_L$.

5 Параметризованные типы

В §2 мы определили для функции $f \in A \to B$ функцию-отображение $f_* \in A_* \to B_*$. Существуют два закона для $_*$: $id_* = id$ и $(f \circ g)_* = f_* \circ g_*$. Эти два закона констатируют, что $_*$ — функтор. Другое характерное свойство отображения в том, что оно не изменяет "форму" аргумента. Отсюда следует, что любой *параметризованный* тип данных имеет таким функтором-отображением. Параметризованный тип — это тип, являющийся наименьшей неподвижной точкой частично примененного бифунктора. В противоположность подходу Малькома [17] отображения могут быть заданы и как катаморфизмы, и как анаморфизмы.

Отображения

Пусть $\dagger$ — бифунктор, тогда мы определям функтор $_*$ на объектах A как параметризованный тип A_*, где $(A_*, in) = \mu(A\dagger)$, а на функциях $f \in A \to B$ как

$$f_* = (\!|in \circ (f\dagger)|\!)_{(A\dagger)} \tag{34}$$

Поскольку $(f\dagger) \in (A\dagger) \to (B\dagger)$, из (27) сразу же получаем альтернативную версию f_* в виде анаморфизма:

$$f_* = [\![(f\dagger) \circ out]\!]_{(B\dagger)}$$

Принадлежность f_* к функторам доказывается следующим образом:

$$\begin{array}{ll} & f_* \circ g_* \\ = & \quad \text{определение } _* \\ & (\!|in \circ (f\dagger)|\!) \circ (\!|in \circ (g\dagger)|\!) \\ = & \quad (29) \\ & (\!|in \circ (f\dagger) \circ (g\dagger)|\!) \\ = & \quad 9 \\ & (\!|in \circ ((f \circ g)\dagger)|\!) \\ = & \quad \text{определение } _* \\ & (f \circ g)_* \end{array}$$

Отображения обладают свойством сохранять "форму". Если определить $SHAPE = VOID_*$, то $SHAPE \circ f_* = VOID \circ f_* = SHAPE$.

Для списка $(A_*, Nil \triangle Cons) = \mu(A\dagger)$ при $A \dagger X = \mathbf{1}|A||X$ и $f \dagger g = id|f||g$ получаем $f_* = [\![f \dagger id \circ out]\!]$. Из УС катаморфизмов находим, что это выражение соответствует привычному определению функции *map*.

$$\begin{array}{rcl} f_* \circ Nil & = & Nil \\ f_* \circ Cons & = & Cons \circ f||f_* \end{array}$$

Также важными законами об отображениях являются *факторизация* [26] и *продвижение* [4].

$$(\!|\varphi|\!) \circ f_* = (\!|\varphi \circ (f\dagger)|\!) \quad (35)$$

$$f_* \circ [\![\psi]\!] = [\![(f\dagger) \circ \psi]\!] \quad (36)$$

$$(\!| \circ f_* = g \circ (\!|\chi|\!) \Leftarrow g \circ \chi = \varphi \circ f \dagger g \wedge g \text{ строгая} \quad (37)$$

$$f_* \circ [\![\psi]\!] = [\![\xi]\!] \circ g \Leftarrow \xi \circ g = f \dagger g \circ \psi \quad (38)$$

Теперь, когда мы знаем, что $_*$ — это функтор, мы можем видеть, что $in \in \mathbf{1}\dagger_* \to_*$ и $out \in_* \to \mathbf{1}\dagger_*$ — естественные преобразования.

$$\begin{array}{rcl} f_* \circ in & = & in \circ f \dagger f_* \\ out \circ f_* & = & f \dagger f_* \circ out \end{array}$$

Продвижение функции *iterate*

Вспомним функцию $iterate\ f = [\![(\acute{i} \circ id \triangledown f)]\!]$. Следующее правило преобразует алгоритм со сложностью $\mathcal{O}(\setminus^{\in})$ в алгоритм со сложностью $\mathcal{O}(\setminus)$ при условии, что вычисление $g \circ f^n$ занимает n шагов.

$$g_* \circ iterate\ f = iterate\ h \circ g \Leftarrow g \circ f = h \circ g \quad (39)$$

Правило (39) — это прямое следствие из закона продвижения для анаморфизмов (38).

Также мы можем интересно записать *iterate* как циклический список:

$$iterate\ f\ x = \mu(\lambda xs.Cons\ (x, f_*\ xs))$$

и использовать для доказательства (39) теорему слияния неподвижной точки.

Факторизация отображения-свертки (map-reduce)

Тип данных $(A_*, in) = \mu(A\dagger)$ при $A \dagger X = A|X_F$ называется *свободным* $_F$-типом над A. Для свободного типа мы всегда можем записать *строгие* катаморфизмы $(\!|\psi|\!)$ как $(\!|f \vartriangle \varphi|\!)$, принимая $f = \psi \circ \grave{i}$ и $\varphi = \psi \circ \acute{i}$. Для f_* получаем

$$\begin{aligned} f_* &= (\!|in \circ f|id|\!) \\ &= (\!|tau|join \circ f|id|\!) \\ &= (\!|tau \circ f \vartriangle join|\!) \end{aligned}$$

где $tau = in \circ \grave{i}$ и $join = in \circ \acute{i}$.

Если мы определим *свертку* φ как

$$\varphi/ = (\!|id \vartriangle \varphi|\!) \tag{40}$$

то из закона факторизации (35) следует, что катаморфизмы на свободных типах могут быть разделены на отображение и последующую свертку.

$$\begin{aligned} & (\!|f \vartriangle \varphi|\!) \\ = & \\ & (\!|id \vartriangle \varphi \circ f|id|\!) \\ = & \\ & (\!|id \vartriangle \varphi|\!) \circ f_* \\ = & \\ & \varphi/ \circ f_* \end{aligned}$$

Тот факт, что tau и $join$ являются естественными преобразованиями, дает нам правила вычисления для f_* и $\varphi/$ на свободных типах.

$$\begin{array}{rclcrcl} f_* \circ tau &=& tau \circ f & \qquad & \varphi/ \circ tau &=& id \\ f_* \circ join &=& join \circ f_{*F} & & \varphi/ \circ join &=& \varphi \circ (\varphi/)_F \end{array}$$

Ранний Squiggol полностью основывался на факторизации отображения-свертки. Некоторые из этих законов — из старых добрых времен; например, *продвижение свертки* и *продвижение отображения*.

$$\begin{aligned} \varphi/ \circ join/ &= \varphi/ \circ (\varphi/)_* \\ f_* \circ join/ &= join/ \circ f_{**} \end{aligned}$$

Монады

Любой свободный тип порождает монаду [17] (в нотации, описанной выше, $(_*, tau \in_{\mathbf{I}} \to_*, join/ \in_{**} \to_*)$), поскольку:

$$\begin{aligned} join/ \circ tau &= id \\ join/ \circ tau_* &= id \\ join/ \circ join/ &= join/ \circ join/_* \end{aligned}$$

Уодлер в [29] подробно обсуждает концепцию монад и их использования в функциональном программировании.

6 Заключение

Мы описали различные шаблоны рекурсивных определений и представили многие законы, выполняющиеся для рекурсивно определенных функций. Хотя мы и проиллюстрировали законы и рекурсивные операторы на примерах, их полезность для практического вычисления программ может не быть очевидной для каждого читателя. К сожалению, здесь нам не хватит места для более детальных примеров.

Существуют и другие аспекты вычисления программ, нежели просто последовательности комбинаций форм (вроде $(\!|\_|\!)$, $[\![\_]\!]$, $\langle\!|\_|\!\rangle$, $[\![\_,\_]\!]$) и законы о них. Для вычисления сложных программ совершенно точно потребуются высокоуровневые алгоритмические теоремы. Работа, отчет о которой здесь представлен, предоставляет необходимые инструменты для разработки таких теорем. Бёрд начал такую работу для теории списков [3], и успешно.

Другим аспектом вычисления программ является автоматизация. Наш опыт — включая и опыт наших коллег — показывает, что объем формализованных операций много больше, чем в большинстве книг по математике; их можно сравнить по объему с "компьютерной алгеброй", как она реализована в системах типа MACSYMA, Maple, Mathematica и подобных. К счастью, большинство операций легко автоматизируются, больше того, некоторые тождества основываются на естественных преобразованиях. Поэтому, в некоторых случаях достаточно лишь проверки типов. Очевидно, автоматизация очень выгодна и не выглядит слишком сложной для реализации.

В конечном счете мы заметили, что теория категорий предоставляет понятия и концепты, незаменимые для разработки ясной и стройной теории; например, понятия функтора и естественного преобразования. (Специалист в теории категорий, читая эту статью, может заметить несколько других понятий, которые мы потихоньку используем). Без сомнений, много больший объем знаний из теории категорий может быть полезен для вычисления программ; мы лишь в самом начале захватывающей разработки.

Благодарности Многие из представленных здесь результатов для случая SET уже появлялись в многочисленных заметках Алгоритмического Клуба STOP, среди других благодаря Roland Backhouse, Johan Jeuring, Doaitse Swierstra, Lambert Meertens, Nico Verwer и Jaap van der Woude. Graham Hutton предоставил множество полезных замечаний по черновику этой статьи.

Список литературы

[1] Roland Backhouse, Jaap van der Woude, Ed Voermans, and Grant Malcolm. Arelational theory of types. Technical Report ??, TUE, 1991.

[2] Rudolf Berghammer. On the use of composition in transformational programming. Technical Report TUM-I8512, TU München, 1985.

[3] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3-42. Springer Verlag, 1987. Also Technical Monograph PRG-56, Oxford University, October 1986.

[4] Richard Bird. Constructive functional programming. In M. Broy, editor, *Marktobedorf International Summer school on Constructive Methods in Computer Science*, NATO Advanced Science Institute Series. Springer Verlag, 1989.

[5] Richard Bird and Phil Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[6] R. Bos and C. Hemerik. An introduction to the category-theoretic solution of recursive domain equations. Technical Report TRCSN 88/15, Eindhoven University of Technology, October 1988.

[7] Manfred Broy. *Transformation parallel ablaufender Programme.* PhD thesis, TU München, München, 1980.

[8] A. de Bruin and E.P. de Vink. Retractions in comparing Prolog semantics. In *Computer Science in the Netherlands 1989*, pages 71-90. SION, 1989.

[9] Peter de Bruin. Naturalness of polymorphism. Technical Report CS 8916, RUG, 1989.

[10] Maarten Fokkinga. Tupling mutumorphisms. *The Squiggolist*, 1(4), 1989.

[11] Maarten Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer. Translating attribute grammars into catamorphisms. *The Squiggolist*, 2(1), 1991.

[12] Maarten Fokkinga and Erik Meijer. Program calculation properties of contonuous algebras. Technical Report 91-4, CWI, 1991.

[13] C. Gunter, P. Mosses, and D. Scott. Semantic domains and denotational semantics. In *Marktoberdorf International Summer school on Logic, Algebra and Computation*, 1989. to appear in: Handbook of Theoretical Computer Science, North Holland.

[14] Tasuya Hagino. Codatatypes in ML. *Journal of Symbolic Computation*, 8:629-650, 1989.

[15] J. Arsac and Y. Kodratoff. Some techniques for recursion removal. *ACM Toplas*, 4(2):295-322, 1982.

[16] D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: a synthetic approach. *Math. Systems Theory*, 14:97-139, 1981.

[17] Grant Malcolm. *Algebraic Types and Program Transformation.* PhD thesis, University of Groningen, The Netherlands, 1990.

[18] Lambert Meertens. Algorithmics — towards rpogramming as a mathematical activity. In *Proceedings of the CWI symposium on Mathematics and Computer Science*, pages 289-334. North-Holland, 1986.

[19] Lambert Meertens. Paramorphisms. To appear in Formal Aspects of Computing, 1990.

[20] Joh-Jules Ch. Meyer. *Programming calculi based on fixed point transformations: semantics and applications.* PhD thesis, Vrije Universiteit, Amsterdam, 1985.

[21] Ross Paterson. *Reasoning about Functional Programs.* PhD thesis, University of Queensland, Brisbane, 1988.

[22] John C. Reynolds. Types abstraction and parametric polymorphism. In *Information Processing '83.* North Holland, 1983.

[23] David A. Shmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[24] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761-785, November 1982.

[25] Joseph E. Stoy. *Denotational Semantics, The Scott-Strachey Approach to Programming Language Theory*. The MIT press, 1977.

[26] Nico Verwer. Homomorphisms, factorisation and promotion. *The Squiggolist*, 1(3), 1990. Also technical report RUU-CS-90-5, Utrecht University, 1990.

[27] Phil Wadler. Views: A way for pattern matching to cohabit with data abstraction. Technical Report 34, Programming Technology Group, University of Göteburg and Chalmers University of Technology, March 1987.

[28] Philip Wadler. Theorems for free! In *Proc. 1989 ACM Conference on Lisp and Functional Programming*, pages 347-359, 1989.

[29] Philip Wadler. Comprehending monads. In *Proc. 1990 ACM conference of Lisp and Functional Programming*, 1990.

[30] M. Wand. Fixed point contructions in order enriched categories. *Theoretical Computer Science*, 8, 1979.

[31] Hans Zierer. Proggrammierung mit funktionsobjecten: Konstruktive erzeugung semantische bereiche und anwendung auf die partielle auswertung. Technical Report TUM-I8803, TU München, 1988.