Функциональное программирование

Антон Ермилов

13 января 2018 г.

Содержание

1. Теория

1. В *императивном программировании* вычисление описывается в терминах инструкций, изменяющих состояние вычислителя.

Соответственно, состояние изменяется инструкциями присваивания значений изменяемым переменным, инструкции выполняются последовательно, а также в таком подходе допустимы ветвления и циклы (if, for, while).

Выполнение программы — переход вычислителя из начального состояния в конечное с помощью последовательности инструкций.

Такой подход также называется фон-неймановским.

В *функциональном программировании* программа — это выражение, а её выполнение — это вычисление (редукция) этого выражения.

Как следствие, в таком подходе отсутствуют состояния (а значит, нет переменных и присваивания) и циклы. Последовательность выполнения также не важна, поскольку выражения независимы.

Вместо этого в функциональном программировании существуют рекурсия (замена циклам), функции высших порядков (map, foldr) и сопоставление с образцом.

Все функции — чистые, т.е. зависят только от значений передаваемых параметров.

2. λ-исчисление — формальная система, лежащая в основе функционального программирования. Была введена Алонзо Чёрчем для формализации и анализа понятия вычислимости. Разделяют бестиповое и типизированное λ-исчисления.

В λ-исчислении имеется два способа строить выражения: применение (аппликация) и абстракция.

Нотация применения F к X — FX, т.е. F (алгоритм) применяется к X (входные данные), однако явного различия между алгоритмом и данными нет. К примеру, возможно применение FF.

Рассмотрим теперь абстракцию.

Пусть $M \equiv M[x]$ — выражение, (возможно) содержащее x. Тогда абстракция $\lambda x\,.\,M$ обозначает функцию $x \mapsto M[x]$, т.е. каждому x сопоставляется $M[x]$.

Лямбда-абстракция — способ задать неименованную (анонимную) функцию.

Если x в $M[x]$ отсутствует, то $\lambda x\,.\,M$ — константная функция со значением M.

Введём определение β-эквивалентности: $(\lambda x\,.\,M)N =_\beta M[x := N]$, где $M[x := N]$ обозначает подстановку N вместо x в M.

В чистом λ-исчислении нет ничего, кроме применения, абстракции и β-эквивалентности.

Введём определение множества λ-термов:

Множество λ-термов Λ индуктивно строится из переменных $V = \{x, y, z, \ldots\}$ с помощью применения и абстракции.

- $x \in V \implies x \in \Lambda$
- $M,\, N \in \Lambda \implies (M\,N) \in \Lambda$
- $M \in \Lambda,\, x \in V \implies (\lambda x\,.\,M) \in \Lambda$

В абстрактном синтаксисе, $\Lambda ::= V \mid (\Lambda\,\Lambda) \mid (\lambda V\,.\,\Lambda)$.

Также существуют следующие соглашения:

- Внешние скобки при записи термов опускаются.
- Применение левоассоциативно.
- Абстракция правоассоциативна.
- Тело абстракции простирается вправо насколько это возможно: $\lambda x\,.\,M\,N\,K \iff \lambda x\,.\,(M\,N\,K)$

Введём определение свободных и связанных переменных:

Множество $\mathrm{FV}(T)$ свободных (free) переменных в λ-терме T:

- $\mathrm{FV}(x) = \{x\}$
- $\mathrm{FV}(M\,N) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$
- $\mathrm{FV}(\lambda x\,.\,M) = \mathrm{FV}(M) \setminus \{x\}$

Множество $\mathrm{BV}(T)$ связанных (bound) переменных в λ-терме T:

- $\mathrm{BV}(x) = \varnothing$
- $\mathrm{BV}(M\,N) = \mathrm{BV}(M) \cup \mathrm{BV}(N)$
- $\mathrm{BV}(\lambda x\,.\,M) = \mathrm{BV}(M) \cup \{x\}$

M — замкнутый λ-терм, если $\mathrm{FV}(M) = \varnothing$. Множество замкнутых λ-термов обозначается Λ^0.

Имена замкнутых переменных можно (почти) безболезненно переименовывать в том смысле, что β-преобразования таких термов дают один и тот же результат. Термы, отличающиеся только именами связных переменных, называются α-эквивалентными.

Также заметим, что функции нескольких переменных можно записать как последовательность функций от одной переменной. Переход от функции нескольких аргументов к функции, принимающей аргументы “по одному”, называется каррированием.

3. $M[x := N]$ обозначает подстановку N вместо *свободных* вхождений x в M. Правила подстановки:

- $x[x := N] = N$
- $y[x := N] = y$
- $(P\,Q)[x := N] = (P[x := N])(Q[x := N])$
- $(\lambda y\,.\,P)[x := N] = \lambda y\,.\,(P[x := N]),\ \ y \notin \mathrm{FV}(N)$
- $(\lambda x\,.\,P)[x := N] = (\lambda x\,.\,P)$

На примере четвёртого правила рассмотрим следующую неприятность: $(\lambda y\,.\,x\,y)[x := y]$ (здесь $y \in \mathrm{FV}(N)$).

Отсюда вытекает следующее соглашение (соглашение Барендрегта): имена связанных переменных всегда выбираются так, чтобы они отличались от имён свободных переменных. В этом случае можно использовать подстановку без оговорки о свободных и связанных переменных.

Лемма подстановки:

Lm. Пусть M, N, $L \in \Lambda$. Предположим $x \not\equiv y$ и $x \notin \mathrm{FV}(L)$.

Тогда $M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$

Proof. нудная индукция по 7 пунктам.

4. Конверсия — преобразование.

β-конверсия — $\forall\, M, N \in \Lambda : (\lambda x\,.\,M)N =_\beta M[x := N]$

Логические аксиомы и правила:

- $M = M$
- $M = N \implies N = M$
- $M = N,\ N = L \implies M = L$
- $M = N \implies MZ = NZ$
- $M = N \implies ZM = ZN$
- $M = N \implies \lambda x\,.\,M = \lambda x\,.\,N$

Если $M = N$ доказуемо в λ-исчислении, то пишут $\lambda \vdash M = N$.

α-конверсия — $\forall\, M \in \Lambda : \lambda x\,.\,M =_\alpha \lambda y\,.\,M[x := y]$ в предположении, что $y \notin \mathrm{FV}(M)$.

Индексы Де Брауна (De Bruijn) — альтернативный способ записи термов. Формально, связанные переменные не именуются, а индексируется, индекс показывает, сколько лямбд "назад" переменная была связана.

К примеру, $\lambda x\,.\,(\lambda y\,.\,x\,y) \iff \lambda(\lambda\ 2\ 1)$.

При таком представлении все α-эквивалентные термы кодируются одинаково.

η-конверсия — $\lambda x\,.\,M\,x =_\eta M$ в предположении, что $x \notin \mathrm{FV}(M)$.

Смысл такой эквивалентности — эквивалентность применения, т.е. $(\lambda x\,.\,M\,x)N =_\beta MN$.

η-преобразование обеспечивает принцип экстенциональности: две функции считаются экстенционально эквивалентными, если они на любом входе дают одинаковый результат.

В Хаскелле т.н. "бесточечный" стиль записи (без точки приложения, т.е. без явного указания аргументов) основан как раз на η-конверсии.

5. Можно расширить множество λ-термов константами: $\Lambda(\mathbb{C}) ::= \mathbb{C} \mid V \mid \Lambda(\mathbb{C})\,\Lambda(\mathbb{C}) \mid \lambda V\,.\,\Lambda(\mathbb{C})$.

К примеру, $\mathbb{C} = \{\mathbf{true}, \mathbf{false}\}$.

Но нам нужно их уметь использовать, т.е. стоит вести ещё и какие-то функции:

$\mathbb{C} = \{\mathbf{true, false, and, or, not}\}$.

Но чтобы использовать это всё, необходимо задать правила. К примеру, $\mathbf{not\ true} =_\delta \mathbf{false}$ и т.д.

Если на множестве термов X (обычно $X \subset \mathbb{C}$) задана "внешняя" функция $f : X^k \to \Lambda(\mathbb{C})$, то для неё добавляем δ-правило:

1) выбираем незанятую константу δ_f

2) для конкретных $M_1, \ldots, M_k \in X$ добавляем правило сокращения:

$\delta_f\ M_1\ \ldots\ M_k =_\delta f(M_1, \ldots, M_k)$

Как правило, для одной f — не одно правило, а целая схема правил. Т.е. задаём поведение не на одном конкретном входе, а на всех (или почти всех) допустимых.

6. Определим кортежи.

Пара: $\lambda x\,y\,f\,.\,f\,x\,y$

Получение первого элемента: $\lambda p\,.\,p\,K$

Получение второго элемента: $\lambda p\,.\,p\,K_*$

Определим булевы значения.

$\text{tru} = \lambda\, t\ f\, .\, t$

$\text{fls} = \lambda\, t\ f\, .\, f$

$\text{not} = \lambda\, b\, .\, b\ \text{fls}\ \text{tru}$

$\text{if} = \lambda\, c\ t\ f\, .\, c\ t\ f$

$\text{or} = \lambda\, a\ b\, .\, a\ \text{tru}\ b$

$\text{and} = \lambda\, a\ b\, .\, a\ b\ \text{fls}$

7. Определим числа в чистом λ-исчислении, т.н. числа Чёрча.

$0 = \lambda\, s\ z\, .\, z$

$1 = \lambda\, s\ z\, .\, s\ z$

$2 = \lambda\, s\ z\, .\, s\ (s\ z)$

$\ldots$

$n = \lambda\, s\ z\, .\, s\ (s\ \ldots (s\ z)))$

Научимся производить стандартные операции с числами Чёрча:

$\text{succ} = \lambda\, n\ s\, .\, s\ (n\ s\ z)$

$\text{plus} = \lambda\, m\ n\, .\, m\ \text{succ}\ n$

$\text{mult} = \lambda\, m\ n\, .\, m\ (\text{plus}\ n)\ 0$

$\text{pow} = \lambda\, b\ e\, .\, e\ (\text{mult}\ b)\ 1 = \lambda\, b\ e\, .\, e\ b$

8. Схема β-преобразования даёт возможность решать простейшие уравнения на термы.

К примеру, найти F, т.ч. $\forall N, M, L\ \ \lambda \vdash F\ M\ N\ L = M\ L\ (N\ L)$:

(a) $F\ M\ N\ L = M\ L\ (N\ L)$

(b) $F\ M\ N = \lambda\, x\, .\, M\ x\ (N\ x)$

(c) $F\ M = \lambda\, g\, .\, \lambda\, x\, .\, M\ x\ (g\ x)$

(d) $F = \lambda\, f\, .\, \lambda\, g\, .\, \lambda x\, .\, f\ x\ (g\ x)$

Но что делать, если уравнение рекурсивное? К примеру, если $F\ M = M\ F$?

Th. Для любого λ-терма $F \in \Lambda$ существует неподвижная точка $X \in \Lambda$, т.ч. $F\ X = X$.

Доказательство. Введём $W \equiv \lambda\, x\, .\, F\ (x\ x)$ и $X \equiv W\ W$.
Тогда $X \equiv W\ W =_\beta F\ (W\ W) \equiv F\ X$. □

Th. Существует комбинатор неподвижной точки Y, т.ч. $\forall F\ \ F\ (Y\ F) = Y\ F$.

Доказательство. Введём $Y \equiv \lambda\, f\, .\, (\lambda\, x\, .\, f\ (x\ x))\ (\lambda\, x\, .\, f\ (x\ x))$.
Тогда $Y\ F \equiv (\lambda\, x\, .\, F\ (x\ x))\ (\lambda\, x\, .\, F\ (x\ x)) =_\beta F\ ((\lambda\, x\, .\, F\ (x\ x))\ (\lambda\, x\, .\, F\ (x\ x))) \equiv F\ (Y\ F)$. □

Y-комбинатор используется для задания рекурсии. К примеру, определим рекурсивно факториал:

$\text{fac} = \lambda\, n\, .\, \text{if}\ (\text{iszro}\ n)\ 1\ (\text{mult}\ n\ (\text{fac}\ (\text{pred}\ n)))$

Абстрагируемся по fac, получив тем самым функцию fac'. Тогда получим fac = fac' fac.

Отсюда видно, что fac — неподвижная точка для вспомогательной функции fac'.

А значит, fac $= Y$ fac'.

Далее это может быть использовано для вычисления факториала от произвольного числа.

9. Def. Терм вида $(\lambda\, x\,.\, M)\ N$ называется β-редексом.

Def. Терм $M[x := N]$ называется сокращением редекса $(\lambda\, x\,.\, M)\ N$.

Def. Бинарное отношение $\mathcal{R}$ над Λ называется **совместимым**, если $\forall M, N, Z \in \Lambda$: $M\ \mathcal{R}\ N \implies$

- $(Z\ M)\ \mathcal{R}\ (Z\ N)$
- $(M\ Z)\ \mathcal{R}\ (N\ Z)$
- $(\lambda\, x\,.\, M)\ \mathcal{R}\ (\lambda\, x\,.\, N)$

Def. Совместимое отношение эквивалентности называют отношением **конгруэнтности** над Λ.

Def. Совместимое, рефлексивное и транзитивное отношение называют отношением **редукции** над Λ.

Введём одношаговую редукцию.

Def. Бинарное отношение β-редукции за один шаг $\to_\beta$ над Λ:

- $(\lambda\, x\,.\, M)\ N \to_\beta M[x := N]$
- $M \to_\beta N \implies ZM \to_\beta ZN$
- $M \to_\beta N \implies MZ \to_\beta NZ$
- $M \to_\beta N \implies \lambda\, x\,.\, M \to_\beta \lambda\, x\,.\, N$

По определению, такое отношение совместимо.

Введём многошаговую редукцию.

Def. Бинарное отношение β-редукции $\twoheadrightarrow_\beta$ над Λ (индуктивно):

- $M \twoheadrightarrow_\beta M$
- $M \to_\beta N \implies M \twoheadrightarrow_\beta N$
- $M \twoheadrightarrow_\beta N,\ N \twoheadrightarrow_\beta L \implies M \twoheadrightarrow_\beta L$

Такое отношение является транзитивным рефлексивным замыканием $\to_\beta$ и, следовательно, является отношением редукции.

Def. Бинарное отношение $=_\beta$ над Λ (индуктивно):

- $M \twoheadrightarrow_\beta N \implies M =_\beta N$
- $M =_\beta N \implies N =_\beta M$
- $M =_\beta N,\ N =_\beta L \implies M =_\beta L$

Отношение $=_\beta$ является отношением конгруэнтности.

St. $M =_\beta N \iff \lambda \vdash M = N$

Доказательство. Индукция по определениям. □

Формально, два терма связаны отношением $=_\beta$, если они достижимы друг для друга по неориентированным стрелочкам $\to_\beta$.

Def. λ-терм M находится в β-нормальной форме (β-NF), если в нём нет подтермов, являющихся β-редексами.

Def. λ-терм M имеет β-нормальную форму, если существует терм N, т.ч. $M =_\beta N$ и N находится в β-NF.

Rem. Не все термы имеют β-нормальную форму: $\Omega \equiv \omega\,\omega \equiv (\lambda\, x\,.\, x\; x)\;(\lambda\, x\,.\, x\; x)$

Rem. Некоторые термы удлинняются при редукции: $\Omega_3 \equiv \omega_3\,\omega_3 \equiv (\lambda\, x\,.\, x\; x\; x)\;(\lambda\, x\,.\, x\; x\; x)$

Rem. Не все последовательности редукций приводят к β-NF (например, терм $K\; I\; \Omega$).

Def. Редукционный граф $G_\beta(M)$ терма $M \in \Lambda$ — это ориентированный мультиграф с вершинами из $\{N \mid M \twoheadrightarrow_\beta N\}$ и дугами $\to_\beta$.

Rem. Не все редукционные графы конечны (к примеру, Ω_3 и все разрастающиеся термы)

10. Th. (Чёрч-Россер)

Если $M \twoheadrightarrow_\beta N$ и $M \twoheadrightarrow_\beta L$, то существует L, т.ч. $N \twoheadrightarrow_\beta L$ и $K \twoheadrightarrow_\beta L$.

Иначе говоря, β-редукция обладает свойством ромба.

Доказательство. Без доказательства. □

Th. (о существовании общего редукта)

Если $M =_\beta N$, то существует L, т.ч. $M \twoheadrightarrow_\beta L$ и $N \twoheadrightarrow_\beta L$.

Доказательство. Разбор двух случаев.

Если $M \twoheadrightarrow_\beta N$, то $L := N$.

Если $M =_\beta K$ и $K =_\beta N$, то по IH найдутся L_1 и L_2, т.ч. $M \twoheadrightarrow_\beta L_1$ и $K \twoheadrightarrow_\beta L_1$, а также $K \twoheadrightarrow_\beta L_2$ и $N \twoheadrightarrow_\beta L_2$.

Тогда по теореме Чёрча-Россера для K, L_1, L_2 найдётся L, т.ч. $L_1 \twoheadrightarrow_\beta L$ и $L_2 \twoheadrightarrow_\beta L$. □

Cons. Если M имеет N в качестве β-NF, то $M \twoheadrightarrow_\beta N$.

Доказательство. $M =_\beta N \implies$ они имеют общий редукт. Но N — нередуцируем. □

В частности, теперь можем показать, что Ω не имеет β-NF.

Cons. λ-терм имеет не более одной β-NF.

Доказательство. Из полученного по теореме Чёрча-Россера свойства ромба и нередуцируемости термов в β-NF. □

11. Способы редуцировать терм:

- Переменная: v — редукция завершена.
- λ-абстракция: $\lambda\, x\,.\, M$ — редуцируем M.
- Аппликация: $M\; N$ — все варианты отсюда.

Выделяют несколько стратегий редукции:

- Нормальная стратегия: сокращаем редекс $(\lambda\, x\,.\, M)\; N$.

- Аппликативная стратегия: редуцируем отдельно все N_i до их нормальной формы N_i', а затем сокращаем редекс $(\lambda\, x\, .\, M)N'$.

Введём понятие головной нормальной формы (HNF):

Def. Если терм имеет вид $\lambda\, x_1\, \ldots\, x_n\, .\, (\lambda\, z\, .\, M)\, N_1\, \ldots\, N_m$, то редекс $(\lambda\, z\, .\, M)\, N_1$ называется головным редексом.

Def. Терм вида $\lambda\, \overrightarrow{x}\, .\, y\, \overrightarrow{N}$, где y — переменная (возможно, из числа $\overrightarrow{x}$), называется головной нормальной формой (HNF). Иными словами, это терм, в котором отсутствует головной редекс.

Def. Слабой головной нормальной формой (WHNF) называется HNF или λ-абстракция (т.е. не редекс на верхнем уровне).

Выделяют следующие синтаксические категории:

- Не абстракции: NA ::= $v \mid P\, Q$.
- Нормальные формы (не абстракции): NANF ::= $v \mid$ NANF NF.
- Нормальные формы: NF ::= $\lambda\, x\, .$ NF $\mid$ NANF.

Теперь можем описать процесс редукции при нормальной стратегии:

(a) $(\lambda\, x\, .\, M)\, N \to M[x := N]$

(b) NA $\to$ NA' $\implies$ NA $N \to$ NA' N

(c) $N \to N' \implies$ NANF $N \to$ NANF N'

(d) $M \to M' \implies \lambda\, x\, .\, M \to \lambda\, x\, .\, M'$

Нормальная стратегия сокращает самый левый внешний редекс.

Аналогично опишем процесс редукции при аппликативной стратегии:

(a) $(\lambda\, x\, .\, M)$ NF $\to M[x :=$ NF$]$

(b) NA $\to$ NA' $\implies$ NA $N \to$ NA' N

(c) $N \to N' \implies$ NANF $N \to$ NANF N'

(d) $N \to N' \implies (\lambda\, x\, .\, M)\, N \to (\lambda\, x\, .\, M)N'$

(e) $M \to M' \implies \lambda\, x\, .\, M \to \lambda\, x\, .\, M'$

Аппликативная стратегия сокращает самый левый внутренний редекс.

Th. (теорема о нормализации)

Если терм M имеет нормальную форму, то последовательное сокращение самого левого внешнего редекса приводит к этой нормальной форме.

То есть, нормальная стратегия гарантированно нормализует нормализуемое.

Rem. Аппликативная стратегия похожа на стратегию вычислений в большинстве ЯП, т.е. сначала вычисляются аргументы, а затем — сама функция.

Rem. Нормальная стратегия, наоборот, используется в "ленивых" ЯП, таких как Haskell.

Rem. Как правило, нет необходимости доводить редукцию до NF, ограничиваются лишь WHNF. Это позволяет избежать захвата переменной при редуцировании замкнутого терма (захват — процесс, при котором свободные переменные терма s при наивной подстановки в терм t становятся связанными).

В функциональных языках существуют следующие механизмы вызова:

- “вызов по значению” — аппликативный порядок редукций до WHNF
- “вызов по имени” — нормальный порядок редукций до WHNF
- “вызов по необходимости” — “вызов по имени” плюс разделение (аргумент вычисляется только в том случае, если он нужен для вычисления окончательного результата)

12. Научимся вычитать единицу из чисел Чёрча.

Для этого нам понадобятся два вспомогательных терма:

zp = pair 0 0

sp = $\lambda\, p\,.$ pair (snd p) (succ (snd p))

Теперь можем определить вычитание:

pred = $\lambda\, n\,.$ fst (n sp zp)

Обобщим этот случай. Для этого рассмотрим следующие термы:

xz = $\lambda\, x\,.$ pair x 0

fs = $\lambda\, f\ p\,.$ pair (f (fst p) (snd p)) (succ (snd p))

rec = $\lambda\, n\ f\ x\,.$ fst (n (fs f) (xz x))

Формально, n — количество итераций, f — применяемая функция, x — база. В xz x будет храниться пара, в которой первый элемент — результат вычислений, второй — номер итерации.

К примеру, факториал тогда можно выразить так:

fac = $\lambda\, n\,.$ rec n ($\lambda\, x\ y\,.$ mult x (succ y)) (xz 1)

13. **Def.** Типы — синтаксические конструкции, приписываемые термам по определённым правилам. Позволяют избежать различных ошибок при работе программы.

Определим систему типов:

Def. (система в стиле Карри)

Термы те же, что и в бестиповом λ-исчислении, каждый терм обладает множеством различных типов, т.е. типизация — неявная.

Введём просто типизированное λ-исчисление $\lambda_{\to}$.

Def. Множество типов $\mathbb{T}$ системы $\lambda_{\to}$ определяется индуктивно:

$\alpha, \beta, \gamma, \ldots \in \mathbb{T}$ — свободные переменные

$\sigma, \tau \in \mathbb{T} \implies \sigma \to \tau \in T$ — типы пространства функций.

Иначе говоря, $\mathbb{T} ::= \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$, где $\mathbb{V}$ — множество типовых переменных.

Типовая стрелка правоассоциативна.

Как типизировать термы:

- Если терм — переменная, то как угодно.
- Если терм — аппликация $M\ N$, то M должно иметь стрелочный тип $\sigma \to \tau$, N должно иметь тип аргумента σ, а сам результат $M\ N$ должен иметь тип τ.
  Т.е. $M : \sigma \to \tau,\ N : \sigma,\ (M\ N) : \tau$.
- Если терм — абстракция $\lambda\, x\,.\, M$, то его тип должен быть стрелочным ($\sigma \to \tau$), причём тип аргумента должен быть σ, а тип абстракции — τ.

Заметим, что в системе Карри терм $\lambda\, x\,.\, x$ может иметь как тип $\alpha \to \alpha$, так и любой другой стрелочный тип.

Def. Множество предтермов Λ строится из переменных из V с помощью аппликации и абстракции.

- $x \in V \implies x \in \Lambda$
- $M, N \in \Lambda \implies (M\ N) \in \Lambda$
- $M \in \Lambda,\ x \in V \implies (\lambda\, x\,.\, M) \in \Lambda$

Иначе говоря, $\Lambda ::= V \mid \Lambda\ \Lambda \mid (\lambda\, V\,.\, \Lambda)$

Def. Утверждение о типизации в $\lambda_{\to}$ «а ля Карри» имеет вид $M : \tau$, где $M \in \Lambda$ и $\tau \in \mathbb{T}$. M иногда называют субъектом, а τ — предикатом.

Def. Объявление — это утверждение о типизации с термовой переменной в качестве субъекта (к примеру, $x : \alpha,\ f : \alpha \to \beta$).

Def. Контекст (базис, окружение) — множество объявлений с различными переменными в качестве субъекта:

$\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$

Контекст можно расширять новыми переменными. Можно рассматривать его как функцию из V в $\mathbb{T}$.

Def. Утверждение $M : \tau$ называется выводимым в контексте Γ ($\Gamma \vdash M : \tau$), если его вывод может быть произведён по правилам:

- $(x : \sigma) \in \Gamma \implies \Gamma \vdash x : \sigma$
- $\Gamma \vdash M : \sigma \to \tau,\ \Gamma \vdash N : \sigma \implies \Gamma \vdash (M\ N) : \tau$
- $\Gamma, x : \sigma \vdash M : \tau \implies \Gamma \vdash (\lambda\, x\,.\, M) : \sigma \to \tau$

Если существуют Γ и τ, т.ч. $\Gamma \vdash M : \tau$, то M называют допустимым термом.

14. Определим систему типов:

Def. (система в стиле Чёрча)

Термы — аннотированные версии бестиповых термов. Каждый терм имеет тип (обычно уникальный), выводимый из способа, которым терм аннотирован, т.е. типизация — явная.

Правила типизации — такие же, как в Карри, только типы указываются везде явно (к примеру, $(\lambda\, x^{\alpha}\,.\, x) : \alpha \to \alpha$).

Def. Множество предтермов $\Lambda_{\mathbb{T}}$ строится из переменных из V с помощью аппликации и аннотированной типами абстракции.

- $x \in V \implies x \in \Lambda_{\mathbb{T}}$
- $M, N \in \Lambda_{\mathbb{T}} \implies (M\ N) \in \Lambda_{\mathbb{T}}$
- $M \in \Lambda_{\mathbb{T}},\ x \in V, \sigma \in \mathbb{T} \implies (\lambda\, x^{\sigma}\,.\, M) \in \Lambda_{\mathbb{T}}$

Иначе говоря, $\Lambda_{\mathbb{T}} ::= V \mid \Lambda_{\mathbb{T}}\ \Lambda_{\mathbb{T}} \mid (\lambda\, V^{\mathbb{T}}\,.\, \Lambda_{\mathbb{T}})$

Def. Утверждение о типизации в $\lambda_{\to}$ «а ля Чёрч» имеет вид $M : \tau$, где $M \in \Lambda_{\mathbb{T}}$ и $\tau \in \mathbb{T}$. M иногда называют субъектом, а τ — предикатом.

Def. Объявление — это утверждение о типизации с термовой переменной в качестве субъекта (к примеру, $x : \alpha,\ f : \alpha \to \beta$).

Def. Контекст (базис, окружение) — множество объявлений с различными переменными в качестве субъекта:

$\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$

Контекст можно расширять новыми переменными. Можно рассматривать его как функцию из V в $\mathbb{T}$.

Def. Утверждение $M : \tau$ называется выводимым в контексте Γ ($\Gamma \vdash M : \tau$), если его вывод может быть произведён по правилам:

- $(x : \sigma) \in \Gamma \implies \Gamma \vdash x : \sigma$
- $\Gamma \vdash M : \sigma \to \tau, \Gamma \vdash N : \sigma \implies \Gamma \vdash (M\ N) : \tau$
- $\Gamma, x : \sigma \vdash M : \tau \implies \Gamma \vdash (\lambda\, x^{\sigma}\, .\, M) : \sigma \to \tau$

Если существуют Γ и τ, т.ч. $\Gamma \vdash M : \tau$, то M называют допустимым термом.

15. Lm. (лемма генерации)

- $\Gamma \vdash x : \sigma \implies (x : \sigma) \in \Gamma$
- $\Gamma \vdash (M\ N) : \tau \implies \exists \sigma : [\Gamma \vdash M : \sigma \to \tau \wedge \Gamma \vdash N : \sigma]$
- $\Gamma \vdash (\lambda\, x\, .\, M) : \rho \implies \exists \sigma, \tau : [\Gamma, x : \sigma \vdash M : \tau \wedge \rho \equiv \sigma \to \tau]$ (а ля Карри)
- $\Gamma \vdash (\lambda\, x^{\sigma}\, .\, M) : \rho \implies \exists \tau : [\Gamma, x : \sigma \vdash M : \tau \wedge \rho \equiv \sigma \to \tau]$ (а ля Чёрч)

Доказательство. Следует из определения выводимости типов в контексте. □

Lm. (о типизируемости подтерма)

Если терм типизируем, то и любой его подтерм типизируем

Lm. Расширение контекста не влияет на выводимость типа.

Lm. Свободные переменные типизируемого терма лежат в контексте.

Lm. Сужение контекста до свободных переменных терма не влияет на выводимость типа.

Rem. Существуют нетипизируемые термы.

Def. Для $\sigma, \tau \in \mathbb{T}$ подстановку τ вместо α в σ обозначим за $[\alpha := \tau]\sigma$

Lm. (о подстановке типа)

$\Gamma \vdash M : \sigma \implies [\alpha := \tau]\Gamma \vdash M : [\alpha := \tau]\sigma$ (а-ля Карри)

$\Gamma \vdash M : \sigma \implies [\alpha := \tau]\Gamma \vdash [\alpha := \tau]M : [\alpha := \tau]\sigma$ (а-ля Чёрч)

Lm. (о подстановке терма)

Подходящая по типу подстановка (вместо некоторой переменной) не меняет типа терма.

Lm. β-редукция не меняет типа терма.

Lm. В системе «а ля Чёрч» терм имеет единственный тип.

Cons. Типизируемые β-конвертируемые термы имеют один и тот же тип в системе «а ля Чёрч».

16. Можно задать стирающее отображение $|\cdot| : \Lambda_{\mathbb{T}} \to \Lambda$, т.ч.

- $|x| = x$
- $|M\ N| = |M|\ |N|$
- $|\lambda\, x\, .\, M| = \lambda\, x\, .\, |M|$

Все атрибутированные типами термы из версии Чёрча “проектируются” в термы в версии Карри. И наоборот, любой терм из версии Карри может быть поднят в термы из версии Чёрча.

Т.о., для произвольного типа $\sigma \in \mathbb{T}$ выполняется, что:

σ обитаем в $\lambda_{\to}$-Карри $\iff$ σ обитаем в $\lambda_{\to}$-Чёрч.

Проблемы разрешимости:

- Соответствует ли заданный терм заданному типу?
- Можно ли вывести тип для заданного терма?
- Обитаем ли заданный тип?

Все эти задачи разрешимы и в системе Карри, и в системе Чёрча.

Def. Терм называют слабо нормализуемым, если сущесвует последовательность редукций, приводящих его к нормальной форме.

Def. Терм называют сильно нормализуемым, если любая последовательность редукций приводит его к нормальной форме.

Def. Систему типов называют слабо нормализуемой если все её допустимые термы слабо нормализуемы.

Def. Систему типов называют сильно нормализуемой если все её допустимые термы сильно нормализуемы.

Th. Системы типов «а ля Чёрч» и «а ля Карри» сильно нормализуемы.

17. Определим правую свёртку следующим образом:

foldr f ini [] = ini

foldr f ini (x : xs) = f x (foldr f ini xs)

Prop. (свойство универсальности)

Если функция g удовлетворяет следующей системе уравнений:

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

то g = foldr f v.

Доказательство — простая индукция. В другую же сторону утверждение и так очевидно.

Смысл свойства в том, что foldr, на самом деле, является единственной такой функцией.

А как протащить через foldr какую-то функцию?

Рассмотрим равенство (h . foldr g w) = foldr f v.

Воспользуемся свойством универсальности.

(h . foldr g w) [] = v

(h . foldr g w) (x : xs) = f x ((h . foldr g w) xs)

Первое равенство нам даёт, что h w = v.

Второе равенство следует из равенства h (g x y) = f x (h y) $\forall$ x, y.

Тогда имеем свойство слияния для foldr:

Prop. (foldr fusion property)

h (g x y) = f x (h y) $\implies$ h . foldr g w = foldr f (h w)

Prop. (свойство слияния для ассоциативного оператора)

Если $\otimes$ — ассоциативный оператор, то ($\otimes$ z) . foldr ($\otimes$) w = foldr ($\otimes$) (w $\otimes$ z).

Доказательство. Положим f = g = ($\otimes$), h = ($\otimes$ z). Тогда в результате преобразований условие h (g x y) = f x (h y) превратится в (x $\otimes$ y) $\otimes$ z = x $\otimes$ (y $\otimes$ z). □

Если же мы рассмотрим равенство foldr g w . map h = foldr f v, то получим ещё одно свойство:

Prop. (foldr-map fusion property)

foldr g w . map h = foldr (g . h) w

18. **Def.** Главным (principle) типом для заданного терма называется такой тип σ, что любой другой тип τ, который можно приписать этому терму, может быть получен с помощью подстановки в σ некоторой типовой переменной.

Def. Подстановка типа — такая операция $S : \mathbb{T} \to \mathbb{T}$, что $S(\sigma \to \tau) = S(\sigma) \to S(\tau)$.

Тождественную подстановку (с пустым носителем) обозначают [].

Подстановка типовых переменных выполняется параллельно.

Def. Композиция двух подстановок — подстановка с носителем, являющимся объединением носителей, над которым последовательно выполнены обе подстановки.

В случае композиции $T \circ S$ мы рассматриваем сначала носитель, который является объединением носителей S и T, а затем последовательно применяем к нему сначала подстановку S, а затем — T.

В частности, подстановки образуют моноид относительно $\circ$ с единицей [].

Def. Унификатором для типов σ и τ называется такая подстановка S, что $S(\sigma) \equiv S(\tau)$.

Def. Унификатор S называется главным унификатором для σ и τ, если для любого другого унификатора S' существует подстановка T, т.ч. $S' = T \circ S$.

19. **Th.** Существует алгоритм унификации U, который для заданных типов σ и τ возвращает:

- Главный унификатор S, если σ и τ могут быть унифицированы.
- Сообщение об ошибке в противном случае

Алгоритм.

- $U(\alpha, \alpha) = []$
- $U(\alpha, \tau) \mid \alpha \in \mathrm{FV}(\tau) =$ ошибка
- $U(\alpha, \tau) \mid \alpha \notin \mathrm{FV}(\tau) = [\alpha := \tau]$
- $U(\tau, \alpha) = U(\alpha, \tau)$
- $U(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) = U(U(\sigma_2, \tau_2)\sigma_1, U(\sigma_2, \tau_2)\tau_1) \circ U(\sigma_2, \tau_2)$

Rem. $U(\sigma, \tau)$ завершается (типы конечны, очевидно)

Rem. $U(\sigma, \tau)$ унифицирует (по индукции)

Rem. $U(\sigma, \tau)$ даёт главный унификатор (по индукции, без доказательства)

20. Прежде чем научиться выводить тип для заданного терма M, научимся строить систему ограничений на типы для M.

Для типизации таких термов необходим контекст Γ, в котором объявляются типы всех свободных переменных.

Def. Для подстановки, унифицирующей систему уравнений $E = \{\sigma_1 \sim \tau_1, \ldots, \sigma_n \sim \tau_n\}$, введём обозначение $S \vDash E$.

Th. Для любых терма $M \in \Lambda$, контекста Γ, $\mathrm{FV}(M) \subseteq \mathrm{dom}\Gamma$, и типа $\sigma \in \mathbb{T}$ существует конечный набор уравнений на типы $E(\Gamma, M, \sigma)$, т.ч. для некоторого подстановки S выполняется:

- $S \vDash E \implies S(\Gamma) \vdash M : S(\sigma)$
- $S(\Gamma) \vdash M : S(\sigma) \implies S' \vDash E$, где S' — некоторая подстановка, которая ведёт себя на типовых переменных из Γ и σ так же, как и S.

Алгоритм.

- $E(\Gamma, x, \sigma) = \{\sigma \sim \Gamma(x)\}$
- $E(\Gamma, M\ N, \sigma) = E(\Gamma, M, \alpha \to \sigma) \cup E(\Gamma, N, \alpha)$
- $E(\Gamma, \lambda\, x\,.\, M, \sigma) = E(\Gamma \cup \{\alpha\}, M, \beta) \cup \{\alpha \to \beta \sim \sigma\}$

Здесь α и β должны быть «чистыми» типовыми переменными.

21. Def. Для $M \in \Lambda$ главной парой назовём пару (Γ, σ), такую что:

- $\Gamma \vdash M : \sigma$
- $\Gamma' \vdash M : \sigma' \implies \exists S : S(\Gamma) \subseteq \Gamma' \wedge S(\sigma) \equiv \sigma'$

Th. (Хиндли-Милнер)

Существует алгоритм PP, возвращающий для $M \in \Lambda$:

- Главную пару, если M имеет тип.
- Ошибку в противном случае.

Пусть $\mathrm{FV}(M) = \{x_1, \ldots, x_n\}, \Gamma_0 = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}, \sigma_0 = \beta$.

Алгоритм.

- $PP(M) \mid U(E(\Gamma_0, M, \sigma_0)) \equiv$ ошибка $=$ ошибка.
- $PP(M) \mid U(E(\Gamma_0, M, \sigma_0)) \equiv S = (S(\Gamma_0), S(\sigma_0))$

Cons. Аналогичные утверждения для замкнутого терма. Разумеется, в этом случае достаточно вывести только главный тип, поскольку контекст отсутствует.

22. Def. Введём тип (), состоящий из всего одного элемента. Обозначим этот тип за 1. Все остальные типы будем далее строить из этого типа при помощи операций сложения, умножения и возведения в степень.

Def. Операцию дизъюнктного сложения будем обозначать за $+$.

Rem. Так, булев тип имеет вид $1 + 1$, а трёхэлементный тип — $1 + 1 + 1$.

Def. Произведение введём как декартово произведение типов и будем обозначать за *.

Rem. 2 * 3 — пары из булева типа и элемента тройки.

Def. Будем говорить, что два типа изоморфны, если между ними существует биекция.

Rem. $2 * 3 \cong 6,\ \ (i_2, j_3) \rightleftarrows (3i + j)_6$

Def. Определим операцию возведения в степень. Тип $Y \hat{} X$ — это функциональный тип $X \to Y$.

Все стандартные алгебраические свойтсва верны для введённых операций.

Def. Введём переменные типа и абстракцию по таким переменным как $\lambda\, X\, .\, T[X]$.

Rem. Для типа "data Maybe x = Nothing | Just x" конструктор типа Maybe записывается как $\lambda\, X\, .\, 1 + X$.

Перейдём к рекурсивным типам. Рассмотрим их на примере списка:

L = 1 + A + A^2 + A^3 + ...= 1 + A * L

A это — в точности определение списка в Haskell: data List a = Nil | Cons a (List a)

Чтобы решить уравнение на типы L = 1 + A * L, требуется ввести комбинатор неподвижной точки Fix для типов.

Тогда L = (λ X . 1 + A * X) L $\implies$ L = Fix (λ X . 1 + A * X).

Для конструкции Fix λ X . T[X] часто используется обозначение μ X . T[X]. В частности, оператор список тогда может быть записан так:

List A = μ X . 1 + A * X

List = λ A . μ X . 1 + A * X

В Haskell можно даже задать явно оператор фиксированной точки типов:

newtype Fix f = In (f (Fix f))

Пример:

data N x = Z | S x

instance Functor N where fmap f Z = Z fmap f (S x) = S (f x)

Можно заметить, что при попытке конструировать таким образом натуральные числа каждое число будет иметь свой тип. А хотим, чтобы все числа имели одинаковый тип. Тогда введём натуральные числа следующим образом:

type Nat = Fix N

23. Научимся копировать рекурсивный тип (зададим тождественное отображение в себя):

copy :: Functor f ⇒ Fix f → Fix f

copy (In x) = In $ fmap copy x

Здесь copy — тождественное преобразование. Можно убедиться на примере с натуральными числами.

Попробуем ввести обобщение copy, которое заменяет упаковку в In :: f (Fix f) → Fix f на некоторую phi :: f a → a.

Получим обобщение понятия свёртки, катаморфизм.

cata :: Functor f ⇒ (f a → a) → Fix f → a

cata phi (In x) = phi $ fmap (cata phi) x

Для данных функтора f и типа a функция phi :: f a → a называется f-алгеброй. Тип a называют носителем.

type Algebra f a = f a → a

cata :: Functor f ⇒ Algebra f a → Fix f → a

Пример f-алгебры:

Рассмотрим всё те же натуральные числа и получим преобразователь из Nat в Int.

phiN :: N Int → Int

phiN Z = 0

phiN (S x) = succ x

Тогда преобразователь будет выглядеть следующим образом:

natToInt :: Nat → Int

natToInt = cata phiN

Также можем заметить, что конструктор In :: f (Fix f) → Fix f тоже является f-алгеброй (с носителем Fix f)

phiIn :: Algebra f (Fix f)

phiIn = In

Эта алгебра также называется инициальной алгеброй. Она сохраняет всю информацию о структуре. Её катаморфизм — тождественное отображение.

copy :: Fix f → Fix f

copy = cata phiIn

24. Введём операцию, обратную к In :: f (Fix f) → Fix f:

out :: Fix f → f (Fix f)

out (In x) = x

Пара из In и out задаёт изоморфизм между Fix f и f (Fix f) (f-изоморфизм)

Рассмотрим очередное преобразование рекурсивного типа в себя:

copy' :: Functor f ⇒ Fix f → Fix f

copy' x = In $ fmap copy' $ out x

Заметим, что это практически то же самое, что и введённый ранее copy.

Напишем обобщение copy', которое заменяет out :: Fix f → f (Fix f) на некоторую psi :: a → f a. Получим анаморфизм:

ana :: Functor f ⇒ (a → f a) → a → Fix f

ana psi x = In $ fmap (ana psi) (psi x)

Для данных функтора f и типа a функция phi :: a → f a называется f-коалгеброй. Тип a называют носителем.

type Coalgebra f a = a → f a

ana :: Functor f ⇒ Coalgebra f a → a → Fix f

Рассмотрим пример с натуральными числами:

psiN :: Coalgebra N Int

psiN 0 = Z

psiN n = S (n - 1)

Тогда можем научится по числу получать его представление в Nat:

intToNat :: Int → Nat

intToNat = ana psiN

Подобно In, функция out является коалгеброй, также называемой терминальной коалгеброй. Её катаморфизм — тождественная функция.

Гилеморфизм — последовательное применение анаморфизма, а затем катаморфизма.

hylo :: Functor f ⇒ Algebra f a → Coalgebra f b → b → a

hylo phi psi = cata phi . ana psi

Ката- и анаморфизм легко выражаются через гилеморфизм:

cata phi = hylo phi out

ana psi = hylo In psi

25. TODO

2. Haskell

1. **Связывание**

   Знак равенство задаёт связывание. При этом первый символ идентификатора должен быть в нижнем регистре.

   Равенство может в т.ч. задавать функцию. В определении функций можно использовать λ-выражения.

   Порядок связываний не важен (с точки зрения кода, не интерпретатора).

   Соглашения об ассоциативности вызовов — такие же как в λ-исчислении.

   Связывание происходит единожды (при повторной попытке возникает ошибка).

   Связывание — ленивое, если явно не указать.

   Рекурсия

   Простой пример — факториал.

   fac n = if n > 1 then n * fac (n - 1) else 1

   Версия с аккумулятором:

   fac n = helper 1 n

   helper acc n = if n > 1 then helper (n * acc) (n - 1) else acc

   Прочие конструкции языка

   Конструкции where и let ...in ...обеспечивают локальное связывание вспомогательных конструкций.

   fac n' = helper 1 n' where

   helper acc n = if n > 1 then helper (n * acc) (n - 1) else acc

   fac n' = let helper acc n = if n > 1 then helper (n * acc) (n - 1) else acc

   in helper 1 n'

   Guards

   В некотором смысле — аналог ветвления, просматривают условия сверху вниз до первого совпадения:

   fac n' = helper 1 n' where

   helper acc n | n > 1 = helper (n * acc) (n - 1)

   | otherwise = 1

2. **Базовые типы**

   - Bool
   - Char
   - Int — целочисленное фиксированного размера
   - Integer — целочисленное произвольного размера
   - type1 → type2 — тип функции
   - (type1, ..., typeN) — кортеж
   - () — юнит, единичный тип с константой ()

type1 — тип списка

Тип Bool представляет собой перечисление:

data Bool = True | False

Здесь Bool — конструктор типа, а True и False — конструкторы данных.

Каррирование и частичное применение

mult a b = (mult a) b (mult a — частично применённая функция)

curry :: ((a, b) → c) → a → b → c — каррирование, переход от функции нескольких аргументов к функции, принимающей аргументы по одному.

Задание типов

Определим комбинатор $k = \ x\ y \to x$.

Такой терм имеет тип $t \to t_1 \to t$. Переменные типа такого терма неявно находятся под квантором всеобщности (т.е. вместо t и t_1 могут быть любые типы).

Классы типов позволяют также наложить ограничения на полиморфный тип:

add :: Num a ⇒ a → a → a

Контекст Num накладывает на тип a определённы ограничения (должны быть определены сложение, умножение и т.п.)

Система модулей

Программа состоит из набора модулей.

Модули позволяют управлять пространствами имён.

Инкапсуляция через списки экспорта и импорта.

```
module A (foo, bar) where
import B (f, g, h)
foo = f g
```

Конфликты имён разрешаются через полные имена:

```
import qualified B (f, g, h)
foo = B.f B.g
```

3. Операторы

Оператор — это комбинация из одного или более символов:

! # $ % & * + . / < = > ? @ ^ | - ~ :

Все операторы бинарные и инфиксные. Исключение: унарный префиксный минус, который всегда ссылается на Prelude.negate.

Операторы могут быть использованы в префиксном стиле: (+) 1 2 ≡ 1 + 2

Функции в свою очередь могут быть использованы в инфиксном стиле: a ‘plus‘ b ≡ a + b

Инфиксные операторы требуют определения:

- приоритета (какой оператор из цепочки выполнять первым)
- ассоциативности (какой оператор из цепочки выполнять первым при равном приоритете)

Пример: infixl 6 *+*, **+**, ‘plus‘

Аппликация имеет наивысший приоритет (10).

Сечения

Операторы — на самом деле, простые функции, а потому допускают частичное применение.

Левое сечение: $(2\ +) \equiv\ x \to 2 + x$

Правое сечение: $(+\ 2) \equiv\ x \to x + 2$

Наличие скобок при задании сечений обязательно.

Бесточечный стиль

В Haskell можно сделать η-редукцию в определении функции.

Пример комбинаторного определения: foo x = bar bas x

Пример бесточечного определения: foo = bar bas

4. Основание

Какие значения принимает Bool?

bot = not bot имеет тип $\bot$ (основание) — значение, разделяемое всеми типами:

$\bot$ :: forall a . a

Ошибкам (не исключениям) также приписывается этот тип.

Ленивость

Haskell гарантирует вызов по необходимости (поведение по умолчанию), т.е. переданные аргументы вычисляются только при обращении к ним.

Функции, игнорирующие значение своего аргумента, называются нестрогими. Для строгих функций, наоборот, всегда выполняется f $\bot$ = $\bot$.

Для форсированного вычисления используется оператор seq :: a → b → b, который интересуется значением первого аргумента.

Однако, если на месте первого аргумента стоит λ-абстракция или структура данных, то вычисления не происходит.

Через seq определяется «энергичная аппликация» $!:

f $! x = seq x (f x)

Когда есть смысл использовать этот оператор? К примеру, в случае факториала с аккумулятором acc из-за ленивости будет содержать в себе thunk — непосчитанное выражение, которое может быть очень длинным. Но можно заставить его считать всё сразу.

5. Сопоставление с образцом

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

Выражение (x, y) представляет собой образец. При вызове swap происходит сопоставление с образцом, при котором, в данном случае, будет проверена корректность конструктора (что передали аргументы нужных типов), после чего переменные x и y будут связаны с заданными значениями.

При наличии нескольких образцов сопоставление происходит сверху вниз, затем слева направо.

Сопоставление может быть:

- успешным
- неудачным
- расходящимся (если ни одно сопоставление не было успешным, при попытке обратиться к undefined и т.п.)

Для сопоставления может также использоваться case ...xs ...:

foo (a, b) = case (a, b) of (1, x) → x (0, y) → 0

Алгебраические типы данных

Перечисление — тип с 0-арными конструкторами данных:

data Color = Red | Green | Blue

Тип-произведение с одним конструктором данных:

data PointDouble = PtD Double Double

Можно параметризовать тип произвольным типовым параметром:

data Point a = Pt a a

Point — оператор над типами, конкретный тип получается его аппликацией к некоторому типу.

Рекурсивные типы: списки.

Встроены, но могли бы быть определены так:

data [] a = [] | a : ([] a)

As-образец

Можем построить псевдоним к образцу:

dup s (x : xs) = x : s

Здесь s — псевдоним, который далее будет восприниматься как (x : xs).

Ленивый образец — образец с тильдой в начале, сопоставление с ним всегда успешно, а вычисление откладывается до момента связывания.

При конструировании типов можно указать, что мы хотим сразу вычислять передаваемые аргументы (сделав строгий конструктор данных):

data Complex a = !a :+ !a
infix 6 :+

6. type и newtype

type задаёт синоним для типа (type String = [Char])

newtype создаёт новый тип с единственным однопараметрическим конструктором.

Метки полей

При создании типов-произведений хочется иметь возможность получать значение из каждого поля.

Для этого можно использовать метки:

data Point a = Pt ptx, pty::a

Метки имеют тип Point a → a и работают как селекторы.

С их помощью можно задать связывание в образце

abs (Pt ptx = x, pty = y) = x * x + y * y

С их помощью можно в т.ч. изменять хранящиеся значения:

p = Pt ptx = 5, pty = 14
p ptx = 42

7. Стандартные функции для работа со списками

tail :: [a] → [a]
take :: Int → [a] → [a]
(++) :: [a] → [a] → [a]
length :: [a] → Int

Функции высших порядков (HOF)

filter :: (a → Bool) → [a] → [a]
map :: (a → b) → [a] → [b]

Бесконечные структуры данных

С помощью рекурсии можно создавать бесконечные списки:

ones = 1 : ones
fives = map (+ 4) ones
take 10 fives

За счёт ленивости это будет работать корректно.

Генерация списков

Имеется компактный способ создавать большие «регулярные» списки.

К примеру, [1..5] = [1,2,3,4,5], [1,3..11] = [1,3,5,7,9,11], [1..] — список всех натуральных чисел.

Для формирования нелинейных последовательностей есть list comprehensions:

[x*x | x ← [1..10]]
$[\,1, 4, 9, 16, 25, 36, 49, 64, 81, 100\,]$

При нескольких генераторах чаще обновляется тот, что правее:

[(x,y,z) | x←[1..19], y←[1..19], z←[1..19], x*x+y*y==z*z]
$[\,(3,4,5), (4,3,5), (5,12,13), (6,8,10), (8,6,10), (8,15,17), (9,12,15), \ldots\,]$

8. Полиморфизм

Параметрический полиморфизм — возможность использовать один и тот же код для значений разных типов.

Специальный полиморфизм (ad hoc) — возможность одной и той же функции задавать разное поведение в зависимости от типа. Реализуется с помощью классов.

Классы типов

Класс типов — это именованный набор имён функций с сигнатурами, параметризованными общим типовым параметром:

class Eq a where
(==) :: a → a → Bool
(/=) :: a → a → Bool

Имя класса типов задаёт ограничение, называемое контекстом:

(==) :: Eq a ⇒ a → a → a

Наследование

Класс Ord наследует все методы класса Eq плюс содержит собственные методы:

class (Eq a) ⇒ Ord a where
(<), (<=), (>=), (>) :: a → a → Bool
max, min :: a → a → a

Допустимо в т.ч. множественное наследование.

Объявление представителей

Тип a является представителем класса типов, если для него реализованы определения функций этого класса (хотя бы минимальный требуемый набор).

В классах типов какие-то функции могут быть реализованы через другие, но эту реализацию можно переопределить.

Тип-представитель может быть в т.ч. полиморфным (можно определить Eq для [a], где a — произвольный полиморфный тип).

При объявлении представителя можно задавать контекст:

instance (Eq a) => Eq [a] where ...

Без указания контекста определение может привести к ошибке при проверке типов.

Стандартные классы типов

Eq — минимальное определение (==).

Ord — минимальное определение compare или (<=).

Enum — минимальное определение toEnum :: Int → a и fromEnum :: a → Int.

Bounded — минимальное определение minBound :: a, maxBound :: a.

Люба красила волосы в ванной комнате ночью. Вчера.

От написания конспекта автору стало плохо на этом месте...

9. Реализация классов типов

Классы типов реализуются через механизм передачи словарей.

Словарь для класса — это запись из его методов:

data Eq a = MkEq {eq,ne::a → a → Bool}

Объявление представителей транслируется в функции, возвращающие этот словарь, или в функции, возвращающие более сложный словарь:

eqInt :: Eq Int
eqInt = MkEq eqInt neInt

Такой словарь можно использовать вместо контекста. Формально, функция будет принимать в качестве первого аргумента теперь этот словарь.

elem :: Eq a → a → [a] → Bool
elem _ _ [] = False
elem e x (y : ys) = eq e x y || elem e x ys

10. Класс типов Num

class (Eq a, Show a) ⇒ Num a where

(+), (-), (*) :: a → a → a

negate :: a → a

abs, signum :: a → a

fromInteger :: Integer → a

Минимальное полное определение — всё, кроме (-) или negate.

Контекста Ord нет — для комплексных, например, он лишний.

У Num есть два наследника: Integral и Fractional.

Integer и Int — представители Integral. Float и Double — представители Fractional.

Автоматического приведения чисел от одного типа к другому в Haskell'е нет.

Рассмотрим также классы типов Show и Read:

Минимальное определение для Show a — show :: a → String или showsPrec :: Int → a → ShowS.

Используется для преобразования выражений в строку.

Аналогично, Read используется для чтения (parse) выражений из строки.

Минимальное определение Read a — readsPrec :: Int → ReadS a, где ReadS a — это синоним для типа String → [(a, String)]. Формально, такая функция возвращает список возможных вариантов парсинга заданной строки.

11. Def. Моноид — это множество с ассоциативной бинарной операцией над ним и нейтральным элементом для этой операции.

class Monoid a where

mempty :: a

mappend :: a → a → a

mconcat :: [a] → a

mconcat = foldr mappend mempty

mempty — нейтральный элемент, mappend — ассоциативная операция.

mappend'у соответствует оператор <>.

Для любого моноида должны выполняться соответствующие ему законы (ассоциативность + единица).

Списки и числа являются моноидами.

12. Правая свёртка

foldr f ini [] = ini
foldr f ini (x : xs) = f x (foldr f ini xs)

Через правую свёртку выражаются многие следующие функции:

sum = foldr (+) 0
prod = foldr (*) 1
concat = foldr (++) []

Левая свёртка

foldl f ini [] = ini
foldl f ini (x : xs) = foldl f (f ini x) xs

Рекурсия хвостовая — оптимизируется. Однако thunk из цепочки вызовов f нарастает.

Строгая версия левой свёртки: foldl'. В таком случае thunk из цепочки вызовов f не нарастает. Такая свёртка является самой эффективной из всех.

Единственная проблема — foldl не умеет работать с бесконечными списками.

Также существуют версии для непустых списков foldl1 и foldr1.

Развёртки

unfoldr — операция, двойственная к свёртке.

unfoldr :: (b → Maybe (a, b)) → b → a

Пример:

unfoldr (x → if x == 0 then Nothing else Maybe (x, x - 1)) → b → [a]

Сканы

Представляют собой списки последовательных шагов свертки.

Для сканов выполняются следующие тождества:

head (scanr f z xs) ≡ foldr f z xs
last (scanl f z xs) ≡ foldl f z xs

13. Класс Foldable

Минимальное полное определение — foldr или foldMap.

class Foldable t where

foldMap :: Monoid m ⇒ (a → m) → t a → m

foldMap f = foldr (mappend . f) mempty

foldr :: (a → b → b) → b → t a → b

foldr f z t = appEndo (foldMap (Endo . f) t) z

Представители класса Foldable: [], Maybe, Set, Map, Tree...

14. Класс Functor позволяет поднять стрелку с уровня типов на уровень контейнеров над типами:

class Functor f where

fmap :: (a → b) → (f a → f b)

<$ :: a → f b → f a

<$ = fmap . const

При этом функция fmap не меняет структуру контейнера. Сам же тип f должен иметь кайнд f :: * → *.

Также существуют следующие операторы:

<$ — функция, определённая для функтора, позволяющая заполнить контейнер одинаковыми значениями.

<$> :: Functor f ⇒ (a → b) → f a → f b

<$> = fmap — оператор для fmap

$> :: Functor

$> = flip (<$) — то же, что и <$, только аргументы в другом порядке.

void :: Functor f ⇒ f a → f ()

void = () <$ — заполняет структуру юнитами.

Если мы хотим определить функтор для Either, (,) и т.п., у которых кайнд * → * → *, то нужно связать их первый параметр с каким-нибудь типом, чтобы можно было объявить функтор:

instance Functor (Either e) where

fmap _ (Left x) = Left x

fmap g (Right y) = Right (g y)

Также все функторы должны подчиняться следующим законам:

- fmap id ≡ id
- fmap (f . g) ≡ fmap f . fmap g

Смысл законов — fmap не должен менять структуру контейнера.

Если мы обозначим за pure операцию поднятия типа на уровень контейнера, то тогда получим следующую свободную теорему для класса Functor:

Th. fmap g . pure ≡ pure . g

15. Уже научились применять стрелку к значениям, завёрнутым в конструктор.

А что делать, если сама стрелка оказалась завёрнута к нему?

app :: f (a → b) → f a → f b

Ответ:

class Functor f ⇒ Applicative f where

pure :: a → f a

<*> :: f (a → b) → f a → f b

Представители класса Applicative:

instance Applicative Maybe where

pure = Just

Nothing <*> _ = Nothing

Just g <*> x = fmap g x

Свяжем Applicative с Functor с помощью следующего закона:

fmap g x ≡ pure g <*> x

Прочие законы для Applicative:

- pure id <*> v ≡ v
- pure g <*> pure x ≡ pure (g x)
- u <*> pure x ≡ pure ($ x) <*> u
- pure (.) <*> u <*> v <*> w ≡ u <*> (v <*> w)

Также для Applicative определены следующие функции:

liftA2 :: Applicative f ⇒ (a → b → c) → f a → f b → f c

liftA2 g a b = g <$> a <*> b

liftA3 :: Applicative f ⇒ (a → b → c → d)→ f a → f b → f c → f d

liftA3 g a b c = g <$> a <*> b <*> c

16. Определим класс Alternative как аппликативный функтор с моноидальной операцией с семантикой сложения.

class Alternative f ⇒ Applicative f where

empty :: f a

(<|>) :: f a → f a → f a

Так, представитель Alternative для списка — в точности то же, что Monoid для списка.

Вспомним теперь про реализацию Monoid для Maybe.

Рассмотрим другую реализацию:

instance Monoid Mayber where

mempty = Nothing

mappend Nothing m = m

mappend m _ = m

В стандартной библиотеке для такой реализации существует упаковка First, возвращающая первый не-Nothing.

Alternative для Maybe является, по сути, упаковкой First.

Законы Alternative:

- (f <|> g) <*> a ≡ (f <*> a) <|> (g <*> a)
- empty <*> a ≡ empty
- f <$> (a <|> b) = (f <$> a) <|> (f <$> b)
- f <$> empty ≡ empty

17. Выберем в качестве определения парсера следующее:

newtype Parser tok a = Parser runParser :: [tok] → Maybe ([tok], a)

Определим функцию проверки корректности для парсера:

satisfy :: (tok → Bool) → Parser tok tok

satisfy pred = Parser f where

f (x : xs) | pred x = Just (xs, x)

f _ = Nothing

Проблема возникает, к примеру при попытке создать Parser Char Int (хотим распарсить цифру).

Рассмотрим тогда Parser как функтор:

digit :: Parser Char Int

digit = digitToInt <$> satisfy isDigit

Если мы заметим, что Parser — композиция трёх функторов ([tok] →, Maybe и (tok,)), то получим следующее определение fmap:

fmap g (Parser p) = Parser $ (fmap . fmap . fmap) g p

Также можем воспринимать Parser как аппликативный функтор. Тогда сможем удобно обрабатывать ошибки и конкатенировать парсеры.

pure x — парсер, который всегда возвращает одно и то же значение x, независимо от входа.

Определим также <*> для двух парсеров как “результат работы первого парсера”, применённый к “результату работы второго парсера на остатке”.

В частности, неудача хотя бы одного из парсеров приведёт к неудаче в целом.

18. Класс типов Traversable

Минимальное полное определение: traverse или sequenceA.

```
class (Functor t, Foldable t) ⇒ Traversable t where
    sequenceA :: Applicative f ⇒ t (f a) → f (t a)
    traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
```

sequenceA — обеспечивает правило коммутации нашего функтора t с аппликативным функтором f.

traverse — своего рода map, проходит по всему внутреннему контейнеру, применяя функцию и коллекционируя эффекты.

Элементарные представители: Maybe, [].

```
instance Traversable Maybe where
    traverse f Nothing = pure Nothing
    traverse f (Just x) = Just <$> f x
instance Traversable [] where
    traverse f [] = pure []
    traverse f (x : xs) = (:) <$> f x <*> traverse f xs
```

Всякий Traversable — это Functor. Более того, при реализованном Traverse возможно задать реализацию Functor «по умолчанию».

Законы Traversable:

- newtype Identity a = Identity {runIdentity::a}
  Тогда traverse Identity ≡ Identity
- traverse (Compose . fmap g2 . g1) ≡ Compose . fmap (traverse g2) . traverse g1
- t . traverse g ≡ traverse (t . g)