1 Бестиповое λ-исчисление

Здесь комбинатор неподвижной точки — стандартный ленивый комбинатор $\mathbf{Y} = \lambda f.(\lambda x.f\,(x\,x))\;(\lambda x.f\,(x\,x))$.

Задачи на редукцию, содержащие комбинатор $\mathbf{Y}$, можно решать двумя способами. Либо тупо раскрывать редукции, либо пользоваться уравнением для неподвижной точки $f\,(\mathbf{Y}\,f) = \mathbf{Y}\,f$. Чтобы доказать, что нормальной формы не существует, достаточно найти бесконечный цикл при нормальном порядке вычислений, то есть такой внешний редекс, который повторно появляется после нескольких внешних бета-редукций.

Решить уравнение в комбинаторах — это построить λ-термы в головной нормальной форме, которые удовлетворяют данному уравнению.

Задачи на решение уравнений в комбинаторах решаются применением подходящего комбинатора неподвижной точки, если одна из частей уравнения имеет вид $\mathbf{M}\;x_1\;x_2\;\ldots\;x_n$. Если ни одна из частей уравнения не имеет такого вида, то в случае, если некоторые x_i заменены на комбинаторы, можно построить такой $\mathbf{M}$, который будет удовлетворять указанному уравнению для всех x_i. Если же обе части уравнений имеют более сложную структуру, тогда придётся использовать другие эвристики.

Чтобы доказать, что уравнение в комбинаторах не имеет решения, можно воспользоваться леммой о генеричности.

Теорема 1. *Пусть M, N — термы λ-исчисления, причем терм M неразрешим (т.е. не имеет нормальной формы), а N имеет н.ф. Тогда для любого контекста $C[\,]$*

$$C[M] \to_\beta N \Rightarrow \forall L(C[L] \to_\beta N)$$

Пользуясь данными указаниями, разберём несколько задач на бестиповое λ-исчисление.

1. Привести к нормальной форме или доказать, что её не существует.

$$\mathbf{Y}\;(\mathbf{Y}\;(\lambda xy.y))\;(\lambda xy.x)$$

   Разберёмся сначала с внутренней редукцией $\mathbf{Y}\;(\lambda xy.y)$. По уравнению, $\mathbf{Y}\;(\lambda xy.y) = (\lambda xy.y)\;(\mathbf{Y}\;(\lambda xy.y))$, что нормализуется к $\lambda y.y$.

Осталось вычислить $\mathbf{Y}\,(\lambda y.y)\,(\lambda xy.x)$. Явно подставив $\lambda y.y$ в $\mathbf{Y}$, получаем терм $(\lambda x.x\,x)\,(\lambda x.x\,x)$, который, как известно, зацикливается.

Данное рассуждение не является полным доказательством отсутствия нормальной формы (поскольку мы сначала сокращали внутренний редекс, чего при нормальном порядке вычислений делать были не должны), но позволяет легко его построить. Обозначим $\mathbf{N} = \mathbf{Y}\,(\lambda xy.y)$ и подставим $\mathbf{N}$ во внешний комбинатор $\mathbf{Y}$, тем самым произведя редукцию согласно нормальному порядку вычислений.

Получим $(\lambda x.\mathbf{N}\,(x\,x))\;(\lambda x.\mathbf{N}\,(x\,x))\;(\lambda xy.x)$. Ещё раз производим β-редукцию внешнего редекса:

$$\mathbf{N}\,((\lambda x.\mathbf{N}\,(x\,x))\,(\lambda x.\mathbf{N}\,(x\,x)))\;(\lambda xy.x)$$

Вот теперь $\mathbf{N}$ — во внешнем редексе, и мы можем воспользоваться тайным знанием, что он нормализуется к $\lambda y.y$. Применение этого терма к первому аргументу возвращает уже знакомый терм $(\lambda x.\mathbf{N}\,(x\,x))\;(\lambda x.\mathbf{N}\,(x\,x))\;(\lambda xy$ с тем же самым внешним редексом.

2. Привести к нормальной форме или доказать, что её не существует:

$$\mathbf{Y}\;((\lambda xyzw.x\,z\,(y\,z)\;w)\;(\lambda xy.x\;(\lambda xyz.y\,z))\;(\lambda z_1z_2.z_1\,z_2)\;(\lambda xy.y))$$

Здесь использование уравнения для неподвижной точки ничего не даст, пока не будет упрощён терм, к которому применяется комбинатор $\mathbf{Y}$. Три внутренних подтерма можно упростить с помощью η-редукции, а именно $\lambda xyzw.x\,z\,(y\,z)\;w$, $\lambda xyz.y\,z$ и $\lambda z_1z_2.z_1\,z_2$. После чего β-редукцией получаем, что аргумент комбинатора $\mathbf{Y}$ — это $\lambda xyz.z$. Далее можно воспользоваться уравнением для неподвижной точки и показать, что нормальная форма достигается.

3. Решить уравнение в комбинаторах или показать, что решения не существует:

$$\forall x(\mathbf{M}\;x = x\;\mathbf{M})$$

Левая часть уравнения имеет требуемый вид. Воспользуемся комбинатором $\mathbf{Y}$ для построения $\mathbf{M}$: $\mathbf{M} = \mathbf{Y}(\lambda mx.x\ m)$. Чтобы привести это выражение к головной нормальной форме, воспользуемся уравнением для $\mathbf{Y}$: $\mathbf{M} = (\lambda mx.x\ m)\ (\mathbf{Y}(\lambda mx.x\ m)) = \lambda x.x\ (\mathbf{Y}(\lambda mx.x\ m))$. Дальше просто подставим $\lambda mx.x\ m$ в $\mathbf{Y}$ и получим:

$$\mathbf{M} = \lambda x_0.x_0\ ((\lambda x_1 y.y\ (x_1\ x_1))\ (\lambda x_1 y.y\ (x_1\ x_1)))$$

4. Решить уравнение в комбинаторах или показать, что решения не существует:

$$\forall x, y(\mathbf{M}\ (x\ y) = x)$$

Здесь пользоваться конструкцией с комбинатором неподвижной точки напрямую нельзя, потому что части уравнения имеют неподходящую структуру. Уравнение выглядит подозрительно, потому что такой комбинатор позволял бы нам отменять применение (вытаскивать любую функцию x из вычисления). Попробуем опровергнуть существование $\mathbf{M}$ с помощью леммы о генеричности. Для этого нужно подобрать хорошее значение x (чтобы x был уже в нормальной форме), но очень неудачный y (зацикливающийся). Проще всего взять $x = \lambda w.w$, $y = (\lambda z.(z\ z))\ (\lambda z.(z\ z))$. Тогда $x\ y$ — это не имеющий нормальной формы терм $(\lambda z.(z\ z))\ (\lambda z.(z\ z))$, и по лемме о генеричности, $\forall V(\mathbf{M}\ V = \lambda w.w)$. Возьмём $V = (\lambda xy.y)\ (\lambda x.x)$, и получим, что $\lambda xy.y = \lambda w.w$. Это противоречивое утверждение, значит, такого комбинатора $\mathbf{M}$ не существует.

2 Типизированное λ-исчисление

Для вывода типа можно просто воспользоваться алгоритмом Хиндли. Если есть уверенность в себе, не обязательно строго придерживаться разбора сверху вниз, можно попробовать типизировать отдельно подтермы, и затем собрать типы воедино.

Всегда выводим тип того терма, который буквально написан в задаче. Никаких редукций предварительно делать не надо, это может привести к ошибочному ответу. Если есть подозрение, что терм не типизируется, достаточно показать, что не типизируется некоторый его подтерм.

Для построения обитателя типа доказываем утверждение в минимальной логике и затем извлекаем терм из доказательства. Задачи на доказательство ненаселённости (в моделях Крипке) здесь не рассматриваются.

1. Вывести тип терма или показать, что его нельзя типизировать:

$$\lambda x_1.\,(\lambda x_2.\,x_2\ (\lambda x_3.\,x_3))\ (\lambda x_4.\,x_4\ (\lambda x_5.\,x_5)\ x_1)$$

   Выводить тип этого терма по Хиндли довольно трудоёмко. Намного проще заметить, что подтермы $\lambda x_3.x_3$, $\lambda x_5.x_5$ и x_1 являются только аргументами функций, поэтому, скорее всего, их типы в общем терме далее уточняться не будут. Типизируем их как $T_3 \to T_3$, $T_5 \to T_5$ и T_1 соответственно.

   Теперь заметим, что терм $\lambda x_4.x_4\ (\lambda x_5.\,x_5)\ x_1$ также является аргументом. Поэтому логично начать типизацию остальных частей общего терма с него. Переменная x_4 применяется к двум аргументам, типы которых уже известны. Положим её тип $(T_5 \to T_5) \to T_1 \to T_4$, тогда аргументом выражения $\lambda x_2.\,x_2\ (\lambda x_3.\,x_3)$ будет выражение типа $((T_5 \to T_5) \to T_1 \to T_4) \to T_4$.

   Теперь типизируем отдельно выражение-функцию $\lambda x_2.\,x_2\ (\lambda x_3.\,x_3)$, для чего строим тип x_2: $(T_3 \to T_3) \to T_2$. Итоговый тип функции: $((T_3 \to T_3) \to T_2) \to T_2$, а всего выражения: $T_1 \to T_2$.

   Осталось унифицировать $((T_5 \to T_5) \to T_1 \to T_4) \to T_4$ и $(T_3 \to T_3) \to T_2$. Сразу же имеем $T_4 = T_2$, $T_3 = T_4$, $T_3 = (T_5 \to T_5) \to T_1$, значит, итоговый тип общего терма: $T_1 \to (T_5 \to T_5) \to T_1$.

2. Доказать населенность типа и построить его обитателя:

$$(A \vee (A \Rightarrow B)) \Rightarrow (((A \Rightarrow B) \Rightarrow A) \Rightarrow A)$$

Внешний конструктор здесь — следование, поэтому каркас вывода (в нотации Фитча) выглядит так:

$*A \vee (A \Rightarrow B)$ вводим x
| Здесь будет остальной вывод
| $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$
$(A \vee (A \Rightarrow B)) \Rightarrow (((A \Rightarrow B) \Rightarrow A) \Rightarrow A)$ $\lambda x.???$

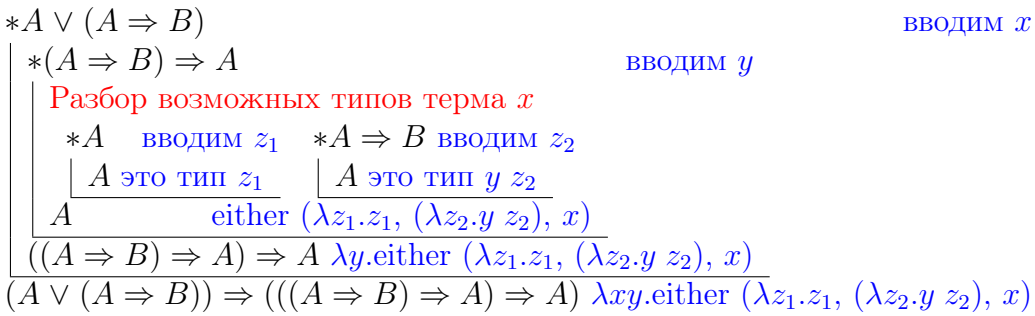
Результат вывода — функция (импликация), поэтому опять нужно применить дедукцию (или, что то же самое, ввести ещё одну абстракцию):

$*A \vee (A \Rightarrow B)$ вводим x
| $*(A \Rightarrow B) \Rightarrow A$ вводим y
| | Осталось только вывести A, имея типы термов x и y
| | A
| $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ $\lambda y.???$
$(A \vee (A \Rightarrow B)) \Rightarrow (((A \Rightarrow B) \Rightarrow A) \Rightarrow A)$ $\lambda xy.???$

Понятно, что придётся разбирать случаи. Сделаем это.

$*A \vee (A \Rightarrow B)$ вводим x
| $*(A \Rightarrow B) \Rightarrow A$ вводим y
| | Разбор возможных типов терма x
| | $*A$ вводим z_1
| | | Выводим A из A
| | | A
| | $*A \Rightarrow B$ вводим z_2
| | | Выводим A из $A \Rightarrow B$ и термов в контексте
| | | A
| | A either $(\lambda z_1.???, \lambda z_2.???, x)$
| $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ $\lambda y.\text{either } (\lambda z_1.???, \lambda z_2.???, x)$
$(A \vee (A \Rightarrow B)) \Rightarrow (((A \Rightarrow B) \Rightarrow A) \Rightarrow A)$ $\lambda xy.\text{either } (\lambda z_1.???, \lambda z_2.???, x)$

В левом подвыводе A выводится переносом из контекста, а в правом — применением импликации (т.е. применением функции). Требуемый терм построен.


$*A \vee (A \Rightarrow B)$
вводим x
$*(A \Rightarrow B) \Rightarrow A$
вводим y
Разбор возможных типов терма x
$*A$
вводим z_1
$*A \Rightarrow B$ вводим z_2
A это тип z_1
A это тип $y\ z_2$
A
either $(\lambda z_1.z_1,\ (\lambda z_2.y\ z_2),\ x)$
$((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ λy.either $(\lambda z_1.z_1,\ (\lambda z_2.y\ z_2),\ x)$
$(A \vee (A \Rightarrow B)) \Rightarrow (((A \Rightarrow B) \Rightarrow A) \Rightarrow A)$ λxy.either $(\lambda z_1.z_1,\ (\lambda z_2.y\ z_2),\ x)$

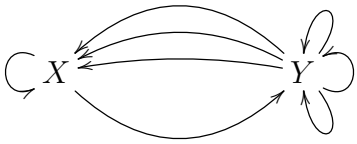
3 Категории

Для того чтобы доказать, что диаграмма описывает категорию, достаточно предъявить алгебраические функции, соответствующие стрелкам диаграммы, и доказать существование всех возможных композиций. Ассоциативность доказывается автоматически по определению композиции для функций. Большинство небольших диаграмм категорий может быть сопоставлено набору конечных множеств и функций между ними. Эвристика «в минимальном множестве, сопоставляемом объекту X, элементов не меньше, чем максимальное число стрелок из одного и того же объекта Y в объект X» (верно и для петель, т.е. стрелок из X в X) работает неплохо, но зачастую можно обойтись и меньшим количеством элементов в множествах модели.

При подозрении, что категория описывается диаграммой некорректно (если отбросить тривиальные случаи, когда в диаграмме очевидно не хватает композиций), доказательство удобно начинать «раскручивать» с тех объектов, у которых нет петель. Если такой объект X существует, и при этом существует путь по стрелкам из X в X, то исходная стрелка точно инъективна (мономорфна), и заключительная — точно сюръективна (эпиморфна). Также имеет смысл обратить внимание на стрелки внутри такого пути, соединящие два объекта Y и Z такие, что других стрелок из Y в Z нет, но есть хотя бы одна петля из Z в Z. Это будет означать, что петля из Z в Z совпадает с единичным морфизмом на области значений стрелки из Y в Z, а значит, эта стрелка не может определять эпиморфизм.

В теории категорий область значений функции f принято называть «кодомен» и обозначать $codom(f)$, двойственным образом к $dom(f)$. В разборе задач в дальнейшем пользуемся этой терминологией.

1. Проверить, соответствует ли категории заданная диаграмма:



   Пусть стрелки из Y в X соответствуют функциям f_1, f_2, f_3, стрелка из X в Y — функции g, петли в Y — функциям h_1, h_2, h_3, а петля в X — функции k.

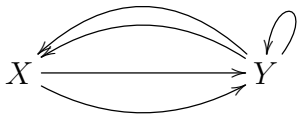
Начнём разбирать диаграмму с композиций $h_i \circ g$. Все они обязаны совпадать с g, поэтому все h_i совпадают с id_Y на $codom(g)$. Значит, все композиции $g \circ f_i$ также совпадают с id_Y на $codom(g)$. Такое малоинтересное поведение функций на данном множестве располагает к тому, чтобы сделать $codom(g)$ как можно меньше. Предположим, что $card(codom(g)) = 1$, то есть $\forall x \in X(g(x) = b_1)$.

Следовательно, $\forall x \in Y(g \circ f_i(x) = b_1)$, и одна из функций h_i обязана быть константой. Без ограничения общности можно взять $h_1(x) = b_1$. Поскольку $f_i \circ h_1$ — это некоторая функция f_j, то она также обязана быть константой. Положим $f_1(x) = a_1$. После этого, очевидно, единственная возможность определить функцию k — это $\forall x \in X(k(x) = a_1)$.

Осталось разобраться с h_2, h_3, f_2, f_3. По построению, $h_i(b_1) = b_1$. Значит, чтобы различить h_2 и h_3, понадобится минимум трёхэлементное множество Y. Положим $h_2(b_2) = b_1$, $h_2(b_3) = b_2$, $h_3(b_2) = b_2$, $h_3(b_3) = b_2$. Тогда $h_2 \circ h_2 = h_1$, $h_2 \circ h_3 = h_1$, $h_3 \circ h_2 = h_2$, $h_3 \circ h_3 = h_2$. Все композиции h_i с h_1 тривиально совпадают с h_1 (это нулевой элемент в полугруппе морфизмов из Y в Y).

Чтобы различить f_i, достаточно положить X двухэлементным (это всё равно необходимо, чтобы k не совпал с единичным морфизмом). Все $f_i(b_1) = a_1$, иначе композиция $f_i \circ g$ не совпадала бы с k. Положим $f_2(b_2) = a_1$, $f_2(b_3) = a_2$, $f_3(b_2) = a_2$, $f_3(b_3) = a_2$, тогда $f_2 \circ h_2 = f_1$, $f_2 \circ h_3 = f_1$, $f_3 \circ h_2 = f_2$, $f_3 \circ h_3 = f_3$. Композиционность соблюдена, категория построена.

2. Проверить, соответствует ли категории заданная диаграмма:



Опять предположим, что стрелки из X в Y задают морфизмы g_i, из Y в X — f_i, из Y в Y — морфизм h. Очевидно, что $\forall i, j(f_i \circ g_j = id_X)$, других возможностей нет. Пусть f_1 и g_1 таковы, что $g_1 \circ f_1 = id_Y$, тогда $g_1 = g_1 \circ (f_1 \circ g_2) = (g_1 \circ f_1) \circ g_2 = g_2$, чего не может быть, поэтому $\forall i, j(g_i \circ f_j = h)$. Тогда $(g_1 \circ f_1) \circ g_2 = h \circ g_2 = g_1$, и $(g_2 \circ f_1) \circ g_2 = h \circ g_2 = g_2$. Это противоречит диаграмме, поэтому категорией она не является.

4 Функторы и естественные преобразования

Задачи на функторы и естественные преобразования даются в контексте пользовательского алгебраического типа данных, который также присутствует в описании задачи вместе с определённым на нём методом <$>. Если в задаче на естественнное преобразование фигурирует функция из контейнерного (функторного) типа в простой, тогда подразумевается, что результатный тип этой функции обёрнут в оболочку `Const`.

```
newtype Const b a = Const { getConst :: b }
instance Functor (Const b) where
        fmap \_ (Const x) = Const x
```

Аргумент типа `a` нужен, чтобы обеспечить параметрический полиморфизм для структуры функтора: если бы его не было, тогда `fmap` был бы обязан брать аргументом функцию типа $b \to b$, а не произвольного типа $a \to a'$ (которая, впрочем, полностью игнорируется применением).

Разберём пример естественного преобразования в простой тип — функция, вычисляющая длину списка.

```
length :: [a] -> Const Int a
length [] = Const 0
length (x:xs) = Const (1 + getConst (length xs))
```

Закон естественного преобразования, применённый к этой функции: $length \circ fmap_{[]} = fmap_{Const} \circ length$. Здесь в нижних индексах указан тип функтора, метод $fmap$ которого рассматривается.

Поскольку $fmap_{[]}$ — это просто map (изменяющий только значения в списке), а $fmap_{Const}$ не меняет целочисленное значение, то $length$ определяет естественное преобразование.

В примерах ниже используется следующий функторный тип.

```
data Log a
    = Var a | ValT | ValF | Or (Log a) (Log a) | Not (Log a)
instance Functor Log where
        fmap f (Var a) = Var (f a)
        fmap f ValT = ValT
        fmap f ValF = ValF
        fmap f (Not m) = Not (fmap f m)
```

```
fmap f (Or m n) = Or (fmap f m) (fmap f n)
```

1. Является ли естественным преобразованием следующая функция?

   ```
   deleteDN (Not (Not x)) = deleteDN x
   deleteDN (Not x) = Not $ deleteDN x
   deleteDN (Or y z) = Or (deleteDN y) (deleteDN z)
   deleteDN x = x
   ```

   Здесь уравнение для естественного преобразования выглядит как $deleteDN \circ fmap_{Log} = fmap_{Log} \circ deleteDN$. Оно очевидным образом выполнено для листьев (структур `ValT`, `ValR` и `Var`), и вообще похоже что выполнено, поскольку функция `deleteDN` оперирует только со структурой функтора, а `fmap` её сохраняет. Попробуем доказать это по индукции.

   Базис есть. Теперь предположим, что для всех деревьев высоты меньше N уравнение выполняется. Рассмотрим древесную структуру высоты N. Возможны три случая.

   (a) Внешний конструктор — `Or`. Тогда

   ```
   deleteDN.(fmap f) $ x `Or` y
       = deleteDN $ (fmap f x) `Or` (fmap f y)
       = (deleteDN.(fmap f) x) `Or` (deleteDN.(fmap f) y).
   ```

   С другой стороны,

   ```
   (fmap f).deleteDN $ x `Or` y
       = (fmap f) $ deleteDN x `Or` deleteDN y
       = ((fmap f).deleteDN x) `Or` ((fmap f).deleteDN y).
   ```

   По индукционному предположению, функции в композиции можно поменять местами.

   (b) Внешний конструктор — `Not`, под которым стоит любой другой конструктор (не `Not`). Тогда

   ```
   deleteDN.(fmap f) $ Not x
       = deleteDN $ Not $ fmap f x
       = Not $ deleteDN.(fmap f) x.
   ```

   С другой стороны,

   ```
   (fmap f).deleteDN $ Not x
   ```

`= (fmap f) $ Not $ deleteDN x`

`= Not $ (fmap f).deleteDN x`.

По индукционному предположению, функции в композиции можно поменять местами.

(c) Третий случай, два вложенных конструктора `Not`, очень похож на предыдущие два, только в нём конструкторы протаскиваются только через `fmap`, чтобы потом быть уничтоженными `deleteDN` (либо уничтожаются им сразу и до `fmap` не доходят, в зависимости от порядка функций в композиции).

Нам удалось доказать, что `deleteDN` — естественное преобразование.

Доказательство естественности преобразования, в целом, не требует высоких идей и тупо проводится по индукции. Единственным тонким моментом здесь является выбор типа индукции: то, что мы выбрали полную индукцию («предположим, что для всех $M < N$..., докажем для N»), а не линейную («предположим, что для N..., докажем для $N + 1$»), позволило единообразно обработать все три индукционных перехода.

2. Является ли естественным преобразованием функция `deleteTaut`? Равенство для логических термов выводится автоматически с помощью подсказки `deriving Eq` (это есть буквальное совпадение двух структур).

```
deleteTaut (Or x y) | triv w v = ValT
                    | otherwise = w `Or` v
                              where w = deleteTaut x
                                    v = deleteTaut y
deleteTaut (Not x) | v == ValT = ValF
                   | v == ValF = ValT
                   | otherwise = Not v
                     where v = deleteTaut x
deleteTaut x = x

triv ValT x = True
triv x ValT = True
```

```
triv x (Not y) | x == y = True
               | otherwise = False
triv (Not x) y | x == y = True
               | otherwise = False
triv _ _ = False
```

Подозрительным здесь является как раз равенство. Хотя `deleteTaut` меняет только структуру дерева, всё-таки она подглядывает и в значения, подвешенные в переменных. Это подозрение позволяет легко построить контрпример: пусть функция `toA` тривиально возвращает литеру `'A'`. Тогда

```
deleteTaut $ fmap toA ((Var 'A') `Or` (Not (Var 'B')))
           = ValT
```

но

```
fmap toA $ deleteTaut ((Var 'A') `Or` (Not (Var 'B')))
           = (Var 'A') `Or` (Not (Var 'A'))
```

Контрпример показывает, что `deleteTaut` естественным преобразованием не является.

3. Задаётся ли определением ниже корректный аппликатив?

```
instance Applicative Log where
   pure x = Var x
   (Var f) <*> y = fmap f y
   (Or f g) <*> y = (f <*> y) `Or` (g <*> y)
   ValT <*> y = ValT
   _ <*> y = ValF
```

Вспоминаем законы аппликативов:

- pure id <*> v = v
- pure f <*> pure x = pure (f x)
- pure(.) <*> u <*> v <*> w = u <*> (v <*> w)
- u <*> pure x = pure (\f -> f x) <*> u

Подозрение вызывает четвёртый закон: слева в нём функциональный аргумент вынуждает нас отбрасывать все оболочки, содержащие, например, конструктор `Not`, поскольку `<*>` в таких случаях возвращает всегда `ValF`, а справа такие оболочки будут обрабатываться в глубину посредством преобразования `fmap`. На этой идее можно построить, например, такой контрпример:

```
(Not $ Var toA) <*> pure 'B' = ValF
    = (Var $ \f -> f 'B') <*> (Not $ Var toA)
    = fmap (\f -> f 'B') (Not $ Var toA)
    = Not $ fmap (\f -> f 'B') (Var toA)
    = Not $ Var $ toA 'B' = Not $ Var 'A'
```

Указанный оператор аппликативом не является. Заметим, что это — достаточно тонкая ошибка. Интерпретатор не видит этой проблемы и успешно компилирует код выше под видом аппликатива.

Доказательство того, что определение задаёт корректный аппликатив (либо функтор), проводится структурной индукцией по терму полностью аналогично доказательству естественности преобразования.

5 Задача на Haskell

Все экзаменационные задачи подразумевают действия над некоторым ADT: его конструкция может либо явно присутствовать в задаче, либо быть частью задачи. Задача может представлять собой вопрос, как должен выглядеть тип и его методы, например, если явно задана функция, которая его обрабатывает, либо как построить или модифицировать некоторую функцию над заданным ADT.

Типичный (впрочем, уже почти разобранный выше) пример: взять ADT для логических формул пропозициональной логики с переменными, константой $True$, а также операциями дизъюнкции и отрицания. Для данного ADT построить функцию удаления тавтологий формы $\Phi \vee \neg \Phi$, где Φ может быть любой формулой. Удаление должно быть исчерпывающим: в итоговом терме не должно быть ни одной такой подформулы.

Заметим, что в задаче ничего не говорится про то, что Φ и $\neg \Phi$ должны быть упрощены (например, не содержать двойных отрицаний), не говорится и о том, что нужно устранять подформулы вида $\neg \Phi \vee \Phi$. Воспользуемся этими послаблениями, чтобы решить задачу максимально быстро (что и требуется сделать на экзамене).

По задаче понятно, что нужно обходить структуру формулы и с помощью автоматически порожденного равенства для нашего ADT проверять в ней все подформулы, удовлетворяющие образцу `(Or v1 (Not v2))` на совпадение `v1` и `v2`. Единственный тонкий момент — `v1` и `v2` должны быть уже упрощены к этому времени (иначе мы пропустим, например, тавтологию $(A \vee (B \vee \neg B)) \vee \neg(A \vee True)$).

Приведём способ разрешения этой проблемы с использованием механизма передачи продолжений (вычислительного контекста).

```
data Log a = Var a | ValT | Not (Log a) | Or (Log a) (Log a)
             deriving (Eq, Show)

elimTaut :: Log Char -> (Log Char -> Log Char) -> Log Char

elimTaut (Or v1 (Not v2)) cont = elimTaut v1 (cont.(check (elimTaut v2 id)))
               where check x1 x2 | x1 == x2 = ValT
                                 | otherwise = x2 `Or` (Not x1)
elimTaut (Or v1 v2) cont = elimTaut v1 (cont.(\v -> v `Or` (elimTaut v2 id)))
elimTaut (Not v) cont = elimTaut v (cont.(\v1 -> Not $ v1))
elimTaut w cont = cont w
```

Функция `elimTaut` принимает вторым аргументом контекст — функцию, которую нужно применить к вычисленной части формулы. Этот контекст накапливается при рекурсивном обходе терма и применяется тогда, когда обход достигает листовой вершины (переменной либо константы). Также мы воспользовались буквальным чтением ТЗ задачи: раз не сказано, что нужно упрощать формулы вида $\neg\,\Phi \vee \Phi$, то никакие другие дизъюнктные формулы, кроме указанной в образце первого предложения, можно не проверять, и сразу подставлять в них результаты упрощений. Обратим внимание на порядок аргументов у вспомогательной функции `check`: он инвертирован относительно исходного порядка вычислений (т.е. `x1` — это упрощённый терм `v2`, а `x2` — упрощённый терм `v1`). Это сделано, чтобы поместить вызов упрощающей функции `check` внутрь функции-продолжения, специализировав его по первому аргументу.

Проверочные задачи: написать вариант решения этой же задачи без использования продолжений; модифицировать вариант с продолжениями так, чтобы подформулы вида $\neg\,\Phi \vee \Phi$ тоже упрощались (но избегая тупого копирования кода).

6 Разное

1. Используя свёртку, но не используя операцию elem и встроенный тип данных Set, определить функцию, удаляющую из списка дубликаты.

2. Частичный порядок Ord задаётся списком пар. Подразумевается, что порядок строгий, т.е. циклов нет. Определить предикат достижимости: C_1 достижима для C_2, если существуют такие D_1, ..., D_n, что $(C_1, D_1) \in Ord$, $(D_n, C_2) \in Ord$ и для всех i $(D_i, D_{i+1}) \in Ord$. Через предикат достижимости определить транзитивное замыкание Ord, используя свёртки.

3. Используя частичное применение (карринг), определить функцию, порождающую по значению Φ и списку $[x_1, \ldots, x_n]$ список пар вида (x_i, Φ).

4. Пусть у предиката достижимости порядок аргументов следующий: C_1, C_2, Ord. Используя частичное применение и этот предикат, но

не используя рекурсивные определения (функции высших порядков использовать можно), построить функцию, принимающую на вход элемент C, порядок Ord и список $[D_1, \ldots, D_n]$ и проверяющую, достижимы ли все элементы D_i из C относительно Ord.