Правила переписывания и немного дефорестации

Функциональные техники оптимизации

Николай Пономарев

Математико-механический факультет СПбГУ

21 декабря 2024 г.

OCaml vs. Haskell

OCaml		Haskell	
`f : 'a`	⇝	`f :: a`	типовые аннотации
`a :: b :: []`	⇝	`a : b : []`	Cons для списков
`[1; 2; 3]`	⇝	`[1, 2, 3]`	списки
`fun x -> body`	⇝	`\x-> body`	λ-функции
`'a list`	⇝	`[a]`	Конструктор типа списка

Чистая ленивость

Гадание на типах

```
(?) :: (b -> c) -> (a -> b) -> a -> c
```

Гадание на типах

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Оптимальная композиция

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

Сколько требуется обходов списка `map f (map g xs)`? Промежуточных списков?

Оптимальная композиция

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

Сколько требуется обходов списка `map f (map g xs)`? Промежуточных списков?

Обходов: 2

Оптимальная композиция

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

Сколько требуется обходов списка `map f (map g xs)`? Промежуточных списков?
Обходов: 2, промежуточных списков: 1

Оптимальная композиция

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

Сколько требуется обходов списка `map f (map g xs)`? Промежуточных списков?
Обходов: 2, промежуточных списков: 1 — плохо!

Оптимальная композиция

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

Сколько требуется обходов списка `map f (map g xs)`? Промежуточных списков?
Обходов: 2, промежуточных списков: 1 — плохо!

Перепишем: `map (f . g) xs`
Потребуется всего 1 обход и 0 промежуточных списков!

Можно оно само?

Хотим автоматически такое делать ⇒ научим компилятор!

Можно оно само?

Хотим автоматически такое делать ⇒ научим компилятор!

```
{-# RULES
"map/map" forall f g xs.
    map f (map g xs) = map (f . g) xs #-}
```

Можно оно само?

Хотим автоматически такое делать ⇒ научим компилятор!

```
{-# RULES
"map/map" forall f g xs.
    map f (map g xs) = map (f . g) xs #-}
```

В GHC правила работают во всём проекте, а мейнтейнеры библиотек могут объявлять свои правила для оптимизации пользовательских функций

Содержательные ограничения

Правила должны иметь вид `f e1 ... en`, при этом `f` не должна стоять под квантором

```
{-# RULES
"let/let" forall f g xs. -- ILLEGAL!
  let { x = let { y = e1 } in e2 } in e3
    = let { y = e1 } in let { x = e2 } in e3 #-}
```

- Правило переписывания ≡ ещё одно определение функции
- Упрощение поиска шаблона
- Не ломает другие оптимизации (inlining, let-floating, beta reduction, case swapping, case elimination, ...)

А минусы будут?

Система настолько проста, что опирается на адекватность автора правил
Нет проверки на

- сохранение семантики
- увеличение производительности
- завершаемость

```
{-# RULES
"commute" forall x y. foo x y = foo y x #-}
```

A, ой

- В стандартной библиотеке есть медленные, зато читаемые функции
- Хотим ускорять (≡ переписывать) и их
- Просто правил переписывания не хватит

```haskell
all :: (a -> Bool) -> [a] -> Bool
all p xs = and (map p xs)
```

```haskell
all' :: (a -> Bool) -> [a] -> Bool
all' p xs = h xs
 where h []       = True
       h (x : xs) = p x && h xs
```

Дефорестация

Определение (Из [MW93])

Deforestation is an automatic transformation scheme for functional programs which attempts to remove unnecessary intermediate data structures

```
case (C2 t1 t2 t3) of
  C1 x1 x2    -> e1
  C2 x1 x2 x3 -> e2
  C3 x1       -> e3
```

⇝

```
let x1 = t1
    x2 = t2
    x3 = t3
in e2
```

Здравая идея, которая в своей полной мощи, оказалась слишком дорогой

foldr всему голова

Оказывается, что через стандартную функцию свёртки foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> [b]
foldr k z []       = z
foldr k z (x : xs) = k x (foldr k z xs)
```

выражается множество функций работы со списками:

```
and xs       = foldr (&&) True xs
map f xs     = foldr (\a b -> f a : b) [] xs
xs ++ ys     = foldr (:) ys xs
concat xs    = foldr (++) [] xs
foldl f z xs = foldr (\b g a -> g (f a b)) id xs z
```

foldr всему голова

Оказывается, что через стандартную функцию свёртки foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> [b]
foldr k z []       = z
foldr k z (x : xs) = k x (foldr k z xs)
```

выражается множество функций работы со списками:

```
and xs       = foldr (&&) True xs
map f xs     = foldr (\a b -> f a : b) [] xs
xs ++ ys     = foldr (:) ys xs
concat xs    = foldr (++) [] xs
foldl f z xs = foldr (\b g a -> g (f a b)) id xs z
```

Можно ли как-то сыграть на этом?

foldr/build fusion

Заведём специальную функцию build

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

и правило переписывания

```
{-# RULES
"foldr/build"
  forall k z (g :: forall b. (a -> b -> b) -> b -> b) .
  foldr k z (build g) = g k z #-}
```

Умный слайд

Теорема (Из [GLP93] на основе [Wad89])

Если для некоторого фиксированного типа A существует

```
g :: forall b. (A -> b -> b) -> b -> b,
```

то выполнено

```
foldr k z (build g) = g k z
```

Да зачем ты нам это всё рассказываешь?!

С помощью foldr и build можно ещё больше разложить функции:

```
[]        = build (\c n -> n)
x : xs    = build (\c n -> c x (foldr c n xs))
map f xs  = build (\c n -> foldr (\a b -> c (f a) b) n xs)
xs ++ ys  = build (\c n -> foldr c (foldr c n ys) xs)
concat xs = build (\c n -> foldr (\x y -> foldr c y x) n xs)
repeat x  = build (\c n -> let r = c x r in r)
```

Да зачем ты нам это всё рассказываешь?!

С помощью foldr и build можно ещё больше разложить функции:

```
[]        = build (\c n -> n)
x : xs    = build (\c n -> c x (foldr c n xs))
map f xs  = build (\c n -> foldr (\a b -> c (f a) b) n xs)
xs ++ ys  = build (\c n -> foldr c (foldr c n ys) xs)
concat xs = build (\c n -> foldr (\x y -> foldr c y x) n xs)
repeat x  = build (\c n -> let r = c x r in r)
```

Вот теперь можно заняться дефорестацией!

Бесконечные списки I/II

```
(||) :: Bool -> Bool -> Bool
True  || _ =  True
False || x =  x

repeat True -- [True, True, ..]

repeat x = build (\c n -> let r = c x r in r)

x = foldr (||) False (repeat True)
```

Вычислим ли x? Какого его значение?

Бесконечные списки I/II

```
(||) :: Bool -> Bool -> Bool
True  || _ =  True
False || x =  x

repeat True -- [True, True, ..]

repeat x = build (\c n -> let r = c x r in r)

x = foldr (||) False (repeat True)
```

Вычислим ли x? Какого его значение?

True!

Бесконечные списки II/II

```
x = foldr (||) False (repeat True) -- repeat <-> build
  = foldr (||) False (build (\c n -> let r = c True r in r)) -- foldr/build
  = (\c n -> let r = c True r in r) (||) False -- beta-reduce
  = let r = (||) True r in r -- beta-reduce
  = True
```

Эффективный all

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and (map p xs)
         = foldr (&&) True (map p xs)
         = foldr (&&) True (build (\c n -> foldr (\a b -> c (p a) b) n xs))
         = (\c n -> foldr (\a b -> c (p a) b) n xs) (&&) True
         = foldr (\a b -> (&&) (p a) b) True xs
         = foldr (\a b -> p a && b) True xs -- inline foldr
         = h xs
 where h [] = True
       h (x : xs) = p x && h xs
```

Обобщенный Cons (Из [PTH01])

```haskell
-- (down 5) = [5, 4, 3, 2, 1]
down :: Int -> [Int]
down v = build (\c n -> down' v c n)

down' 0 cons nil = nil
down' v cons nil = cons v (down' (v - 1) cons nil)

x = sum (down 5)
  = foldr (+) 0 (down 5)
  = foldr (+) 0 (build (\c n -> down' 5 c n))
  = (\c n -> down' 5 c n) (+) 0
  = down' 5 (+) 0
```

Хотим бóльшего (по мотивам [Rei24])

Обобщим

```
{-# RULES
"unzip/map1" forall xs.
  unzip (map (\x -> (x, x)) xs) = (xs, xs) #-}
```

до

```
{-# RULES
"unzip/map3" forall f g xs.
  unzip (map (\x -> (f x, g x)) xs)
    = (map f xs, map g xs) #-}
```

и не сможем помэтчить `unzip (map (\x-> (x, x)) xs)`

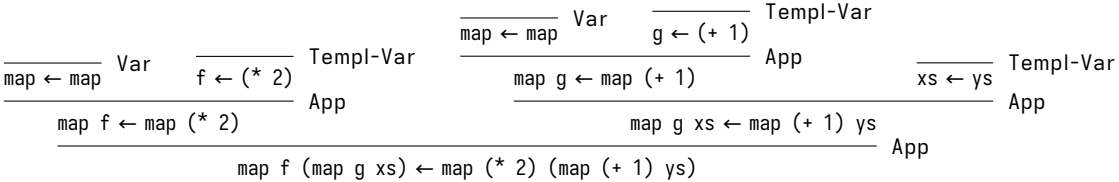
Почему так вышло?

$$\frac{\texttt{f} \leftarrow \texttt{g} \quad \texttt{x} \leftarrow \texttt{y}}{\texttt{f x} \leftarrow \texttt{g y}}\ \texttt{App} \qquad \frac{\texttt{x} \leftarrow \texttt{y[v := u]}}{\texttt{(\u} \rightarrow \texttt{x)} \leftarrow \texttt{(\v} \rightarrow \texttt{y)}}\ \texttt{Lam}$$

$$\frac{}{\texttt{v} \leftarrow \texttt{v}}\ \texttt{Var} \qquad \frac{\texttt{v templ} \quad \mathit{flvs}(\texttt{x}) = \varnothing}{\texttt{v} \leftarrow \texttt{x}}\ \texttt{Templ-Var}$$

$$\frac{\texttt{v unfold to y} \quad \texttt{x} \leftarrow \texttt{y}}{\texttt{x} \leftarrow \texttt{v}}\ \texttt{Var-Unfold} \qquad \frac{\mathit{flvs}(\texttt{v}) = \varnothing \quad \texttt{x} \leftarrow \texttt{z}}{\texttt{x} \leftarrow \texttt{let v = y in z}}\ \texttt{Let-Float}$$

Пример вывода



Чиним!

Интуитивно понятно, что

```
unzip (map (\x → (f x, g x)) xs) ← unzip (map (\x → (x, x)) xs)
```

только в случае $[\mathtt{f} := (\backslash \mathtt{y} \to \mathtt{y}), \mathtt{g} := (\backslash \mathtt{z} \to \mathtt{z})]$

Добавим новое правило!

$$\frac{\begin{array}{l} \mathtt{v}_1 \ldots \mathtt{v}_n \text{ local} \quad \mathtt{v}_1 \ldots \mathtt{v}_n \text{ distinct} \\ \mathtt{f} \text{ templ} \quad \mathit{flvs}(\mathtt{x}) \subseteq \{\mathtt{v}_1, \ \ldots, \ \mathtt{v}_n\} \end{array}}{\mathtt{f}\ \mathtt{v}_1 \ldots \mathtt{v}_n \leftarrow \mathtt{x}}\ \text{HOP}$$

Оно генерирует подстановку $[\mathtt{f} := (\backslash \mathtt{v}_1 \ldots \mathtt{v}_n \to \mathtt{x})]$

Вывод с новым правилом

```
{-# RULES "foo" forall f. foo (\y -> f y) = "RULE FIRED" #-}
```

$$\dfrac{\dfrac{}{\texttt{foo} \leftarrow \texttt{foo}}\,\text{Var} \qquad \dfrac{\dfrac{\texttt{y local} \quad \texttt{y distinct} \quad \texttt{f templ} \quad \mathit{flvs}(\texttt{(+) ((*) y 2) y}) = \{\texttt{y}\} \subseteq \{\texttt{y}\}}{\texttt{f y} \leftarrow \texttt{(+) ((*) y 2) y}}\,\text{HOP}}{\texttt{(\y -> f y)} \leftarrow \texttt{(\x -> (+) ((*) x 2) x)}}\,\text{Lam}}{\texttt{foo (\y -> f y)} \leftarrow \texttt{foo (\x -> (+) ((*) x 2) x)}}\,\text{App}$$

Stream fusion (по мотивам [FHG14]) I/III

Идея: заменить последовательность рекурсивных «трансформеров» на последовательность нерекурсивных «трансформеров», которую завершает рекурсивный «вычислитель»

```
data Stream a where
  Stream :: (s -> Step a s) -> s -> Stream a

data Step a s = Yield a s | Skip s | Done
```

Stream fusion II/III

```haskell
stream :: [a] -> Stream a
stream xs = Stream uncons xs
  where uncons :: [a] -> Step a [a]
        uncons []       = Done
        uncons (x : xs) = Yield x xs
```

```haskell
unstream :: Stream a -> [a]
unstream (Stream g s) = go s
  where go s = case g s of
                 Done       -> []
                 Skip s'    -> go s'
                 Yield x s' -> x : go s'
```

```haskell
mapS :: (a -> b) -> Stream a -> Stream b
mapS f (Stream g s0) = Stream mapStep s0
  where mapStep s = case g s of
                      Done       -> Done
                      Skip s'    -> Skip s'
                      Yield x s' -> Yield (f x) s'
```

```haskell
map :: (a -> b) -> [a] -> [b]
map f = unstream . mapS f . stream
```

Stream fusion III/III

Основа оптимизации — правило:

```
{-# RULES "stream/unstream"  (.) stream unstream = id #-}
```

тогда

```
x = map f . map g
  = unstream . mapS f . stream . unstream . mapS g . stream
  = unstream . mapS f . mapS g . stream -- opts
  = let go []       = []
        go (x : xs) = f (g x) : go xs
    in go
```

А вот и не fusion

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = unstream . concatMapS (stream . f) . stream

concatMapS :: (a -> Stream b) -> Stream a -> Stream b
concatMapS f (Stream g s) = Stream g' (s, Nothing)
 where
  g' (s, Nothing) = case g s of
      Done       -> Done
      Skip s'    -> Skip (s', Nothing)
      Yield x s' -> Skip (s', Just (f x))
  g' (s, Just (Stream g'' s'')) = case g'' s'' of
      Done       -> Skip    (s, Nothing)
      Skip s'    -> Skip    (s, Just (Stream g'' s'))
      Yield x s' -> Yield x (s, Just (Stream g'' s'))
```

Слишком сложная функция

The inner stream, including its generator function, is created by applying a function to a value of the outer stream at runtime. That function could potentially pick from arbitrarily many different inner streams based on the value it is applied to. Each of these streams may have an entirely different generator function. In fact, since the type of the state in a Stream is existentially quantified, the returned streams may not even have the same state type. ([FHG14])

Но обычно всё хорошо

Однако обычно вид функции не зависит от входного элемента. В таком случае можно использовать альтернативную функцию

```
concatMapS' :: (a -> s -> Step s b) -> (a -> s) -> Stream a -> Stream b
```

и правило переписывания

```
{-# RULES
"concatMapS" forall next f.
  concatMapS (\x -> Stream (next x) (f x))
    = concatMapS' next f #-}
```

Поскольку next и f — функции, то тут пригодится HOP правило

We're stuck!

`map (\x-> x + 1) ys` перепишется как `foldr (\x xs -> (x + 1) : xs) [] ys`

Мы застряли!

Можно выбрать

заинлайнить `foldr` и упростить:

```
let go [] = []
    go (x : xs) = (x + 1) : go xs
in go ys
```

GHC выбирает второй вариант

или вернуться к

`map (\x-> x + 1) ys`

Откат переписывания

Без HOP выписать общее правило не получится, поэтому GHC использует функции-маркеры

```
{-# RULES "map" forall f xs.
  map f xs = build (\c n -> foldr (mapFB c f) n xs) #-}

{-# RULES "mapList" forall f.
  foldr (mapFB (:) f) [] = map f #-}
```

С HOP правилом можно избавиться от маркеров и напрямую заменять

```
{-# RULES "mapList2" forall f.
  foldr (\x xs -> f x : xs) [] = map f #-}
```

Источники I

[FHG14] Andrew Farmer, Christian Hoener zu Siederdissen и Andy Gill. «The HERMIT in the stream: fusing stream fusion's concatMap». В: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. PEPM '14. New York, NY, USA: Association for Computing Machinery, 11 янв. 2014, с. 97—108. ISBN: 978-1-4503-2619-3. DOI: 10.1145/2543728.2543736. URL: https://doi.org/10.1145/2543728.2543736 (дата обр. 19.12.2024).

[GLP93] Andrew Gill, John Launchbury и Simon L. Peyton Jones. «A short cut to deforestation». В: *Proceedings of the conference on Functional programming languages and computer architecture*. FPCA '93. New York, NY, USA: Association for Computing Machinery, 1 июля 1993, с. 223—232. ISBN: 978-0-89791-595-3. DOI: 10.1145/165180.165214. URL: https://dl.acm.org/doi/10.1145/165180.165214 (дата обр. 29.09.2024).

[MW93] Simon Marlow и Philip Wadler. «Deforestation for Higher-Order Functions». В: *Functional Programming, Glasgow 1992*. Под ред. John Launchbury и Patrick Sansom. London: Springer, 1993, с. 154—165. ISBN: 978-1-4471-3215-8. DOI: 10.1007/978-1-4471-3215-8_14.

Источники II

[PTH01] Simon Peyton Jones, Andrew Tolmach и Tony Hoare. «Playing by the Rules: Rewriting as a practical optimisation technique in GHC». В: *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop (HW'2001)*. Firenze, Italy, 2 сент. 2001, с. 209—239.

[Rei24] Jaro Reinders. «Higher Order Patterns for Rewrite Rules». В: *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium*. Haskell '24: 17th ACM SIGPLAN International Haskell Symposium. Milan Italy: ACM, 29 авг. 2024, с. 14—26. ISBN: 979-8-4007-1102-2. DOI: 10.1145/3677999.3678275. URL: https://dl.acm.org/doi/10.1145/3677999.3678275 (дата обр. 11.12.2024).

[Wad89] Philip Wadler. «Theorems for free!» В: *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*. the fourth international conference. Imperial College, London, United Kingdom: ACM Press, 1989, с. 347—359. ISBN: 978-0-89791-328-7. DOI: 10.1145/99370.99404. URL: http://portal.acm.org/citation.cfm?doid=99370.99404 (дата обр. 25.11.2024).

Вопросы к экзамену

1. Правила переписывания как способ оптимизации. Мотивационный пример. Необходимые условия корректности. Вид левой части правил в GHC, причины такого выбора. «Законы» правил переписывания в GHC.
2. Используя foldr/build fusion устраните промежуточные структуры данных в выданной функции. (Практическая задачка, я подразумеваю, что слайды 10 и 13 будут в доступе на экзамене.)
3. Постройте вывод в исчислении со слайда 19 для выданной функции и шаблона. (Практическая задачка, я подразумеваю, что слайд 19 будет в доступе на экзамене.)