# Функциональное программирование

Лекция 4

Степан Львович Кузнецов

НИУ ВШЭ, факультет компьютерных наук

# Типизация и рекурсия

- Для простого типизованного $\lambda$-исчисления ($\lambda_{\to}$) по Карри имеется алгоритм вычисления наиболее общего типа.

## Типизация и рекурсия

- Для простого типизованного $\lambda$-исчисления ($\lambda_{\to}$) по Карри имеется алгоритм вычисления наиболее общего типа.
- Однако комбинатор $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ нетипизируем в системе $\lambda_{\to}$.

# Типизация и рекурсия

- Для простого типизованного $\lambda$-исчисления ($\lambda_\to$) по Карри имеется алгоритм вычисления наиболее общего типа.
- Однако комбинатор $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ нетипизируем в системе $\lambda_\to$.
- Чтобы восстановить рекурсию, можно добавить константу $\mathbb{Y}$ с редукцией $\mathbb{Y}u \to_\delta u(\mathbb{Y}u)$ и полиморфным типом $(r \to r) \to r$.

## Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст $\Gamma$.

# Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст $\Gamma$.
- Значит, входящая в него переменная станет неизменяемой, $\mathbb{Y}_p : (p \to p) \to p$.

# Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст $\Gamma$.
- Значит, входящая в него переменная станет неизменяемой, $\mathbb{Y}_p : (p \to p) \to p$.
- Это плохо: нам *«не хватает полиморфизма»*, чтобы типизовать $\mathbb{Y}$.

# Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст $\Gamma$.
- Значит, входящая в него переменная станет неизменяемой, $\mathbb{Y}_p : (p \to p) \to p$.
- Это плохо: нам *«не хватает полиморфизма»*, чтобы типизовать $\mathbb{Y}$.
- Можно обойти эту проблему, использовав конструктор термов Y:

$$\frac{\Gamma, x : r_2 \vdash u : r_3}{\Gamma \vdash (\mathrm{Y}x.u) : r_1}\ \mathrm{Fix} \qquad r_1 \approx r_2 \approx r_3 \qquad \mathrm{Y}x.u \to_\delta u[x := \mathrm{Y}x.u]$$

# Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст $\Gamma$.
- Значит, входящая в него переменная станет неизменяемой, $\mathbb{Y}_p : (p \to p) \to p$.
- Это плохо: нам *«не хватает полиморфизма»*, чтобы типизовать $\mathbb{Y}$.
- Можно обойти эту проблему, использовав конструктор термов $\mathrm{Y}$:

$$\frac{\Gamma, x : r_2 \vdash u : r_3}{\Gamma \vdash (\mathrm{Y}x.u) : r_1}\ \mathrm{Fix} \qquad r_1 \approx r_2 \approx r_3 \qquad \mathrm{Y}x.u \to_\delta u[x := \mathrm{Y}x.u]$$

- Безопасность типов при $\delta$-редукции соблюдается.

## Квантор ∀ по типам

- Неявно в полиморфной типизации по Карри присутствуют *кванторы всеобщности* по переменным $r_j$:

$$f : (p \to p) \vdash \lambda g.\lambda z.f(gz) : \forall r.((r \to p) \to r \to p).$$

# Квантор ∀ по типам

- Неявно в полиморфной типизации по Карри присутствуют *кванторы всеобщности* по переменным $r_j$:

$$f : (p \to p) \vdash \lambda g.\lambda z.f(gz) : \forall r.((r \to p) \to r \to p).$$

- По смыслу, $\mathbb{Y} : \forall r.((r \to r) \to r)$, и мы хотим поместить эту декларацию в контекст (чего нельзя сделать в $\lambda_\to$).

# Квантор ∀ по типам

- Неявно в полиморфной типизации по Карри присутствуют *кванторы всеобщности* по переменным $r_j$:

$$f : (p \to p) \vdash \lambda g.\lambda z.f(gz) : \forall r.((r \to p) \to r \to p).$$

- По смыслу, $\mathbb{Y} : \forall r.((r \to r) \to r)$, и мы хотим поместить эту декларацию в контекст (чего нельзя сделать в $\lambda_{\to}$).
- Употребление в контексте типов с кванторами ∀ *на внешнем уровне* разрешается в *системе типов Хиндли – Милнера.*

## Квантор ∀ по типам

- Неявно в полиморфной типизации по Карри присутствуют *кванторы всеобщности* по переменным $r_j$:

$$f : (p \to p) \vdash \lambda g.\lambda z.f(gz) : \forall r.((r \to p) \to r \to p).$$

- По смыслу, $\mathbb{Y} : \forall r.((r \to r) \to r)$, и мы хотим поместить эту декларацию в контекст (чего нельзя сделать в $\lambda_{\to}$).
- Употребление в контексте типов с кванторами ∀ *на внешнем уровне* разрешается в *системе типов Хиндли – Милнера.*
- Однако мы сначала познакомимся с *системой F*, или $\lambda 2$ (типизованное $\lambda$-исчисление второго порядка), где квантор разрешается использовать вообще без ограничений.

# Система F

- Как и $\lambda_{\to}$, система F — это система типов поверх обычного $\lambda$-исчисления.

# Система F

- Как и $\lambda_{\to}$, система F — это система типов поверх обычного $\lambda$-исчисления.
- Типы строятся из базовых типов (переменных $r_j$ и констант $p_i$) с помощью двух конструкций: $A \to B$ и $\forall r.A$.

# Система F

- Как и $\lambda_{\to}$, система F — это система типов поверх обычного $\lambda$-исчисления.
- Типы строятся из базовых типов (переменных $r_j$ и констант $p_i$) с помощью двух конструкций: $A \to B$ и $\forall r.A$.
- Правила типизации (по Карри):

$$\frac{}{\Gamma, x : A \vdash x : A}\ \text{Ax} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash (\lambda x.u) : (A \to B)}\ \text{Abs}$$

$$\frac{\Gamma \vdash u : (A \to B) \quad \Gamma \vdash v : A}{\Gamma \vdash (uv) : B}\ \text{App}$$

$$\frac{\Gamma \vdash u : A}{\Gamma \vdash u : (\forall r.A)}\ \text{Gen} \qquad \frac{\Gamma \vdash u : (\forall r.B)}{\Gamma \vdash u : B[r := A]}\ \text{Inst}$$

# Система F

- С помощью ∀ можно типизовать применение функции к самой себе: например, $x : \forall r.(r \to r) \vdash (xx) : \forall r.(r \to r)$.

# Система F

- С помощью $\forall$ можно типизовать применение функции к самой себе: например, $x : \forall r.(r \to r) \vdash (xx) : \forall r.(r \to r)$.

$$\dfrac{\dfrac{x : \forall r.(r \to r) \vdash x : \forall r.(r \to r)}{x : \forall r.(r \to r) \vdash x : (r \to r) \to (r \to r)}\ \text{Inst} \qquad \dfrac{x : \forall r.(r \to r) \vdash x : \forall r.(r \to r)}{x : \forall r.(r \to r) \vdash x : r \to r}\ \text{Inst}}{\dfrac{x : \forall r.(r \to r) \vdash xx : r \to r}{x : \forall r.(r \to r) \vdash xx : \forall r.(r \to r)}\ \text{Gen}}\ \text{App}$$

## Система F

- С помощью $\forall$ можно типизовать применение функции к самой себе: например, $x : \forall r.(r \to r) \vdash (xx) : \forall r.(r \to r)$.

$$\dfrac{\dfrac{x : \forall r.(r \to r) \vdash x : \forall r.(r \to r)}{x : \forall r.(r \to r) \vdash x : (r \to r) \to (r \to r)}\ \text{Inst} \qquad \dfrac{x : \forall r.(r \to r) \vdash x : \forall r.(r \to r)}{x : \forall r.(r \to r) \vdash x : r \to r}\ \text{Inst}}{\dfrac{x : \forall r.(r \to r) \vdash xx : r \to r}{x : \forall r.(r \to r) \vdash xx : \forall r.(r \to r)}\ \text{Gen}}\ \text{App}$$

- Далее, можно применить $\lambda$-абстракцию:

$$\vdash \lambda x.(xx) : \big(\forall r.(r \to r)\big) \to \forall r.(r \to r).$$

# Система F

- С помощью $\forall$ можно типизовать применение функции к самой себе: например, $x : \forall r.(r \to r) \vdash (xx) : \forall r.(r \to r)$.

$$\dfrac{\dfrac{x : \forall r.(r \to r) \vdash x : \forall r.(r \to r)}{x : \forall r.(r \to r) \vdash x : (r \to r) \to (r \to r)}\ \text{Inst} \qquad \dfrac{x : \forall r.(r \to r) \vdash x : \forall r.(r \to r)}{x : \forall r.(r \to r) \vdash x : r \to r}\ \text{Inst}}{\dfrac{x : \forall r.(r \to r) \vdash xx : r \to r}{x : \forall r.(r \to r) \vdash xx : \forall r.(r \to r)}\ \text{Gen}}\ \text{App}$$

- Далее, можно применить $\lambda$-абстракцию:

$$\vdash \lambda x.(xx) : \big(\forall r.(r \to r)\big) \to \forall r.(r \to r).$$

- Тем не менее, все типы, типизуемые в системе F, обладают свойством сильной нормализуемости [J.-Y. Girard], поэтому типизовать $\Omega = (\lambda x.(xx))(\lambda x.(xx))$ не получится.

# Система F в Haskell'е

- В Haskell'е система F включается прагмой `RankNTypes`.

## Система F в Haskell'е

- В Haskell'е система F включается прагмой `RankNTypes`.
- Пример:

```
applyToTuple = (\f -> \(x, y) -> (f x, f y))
  :: (forall a. [a] -> b) -> ([c], [d]) -> (b,b)
applyToTuple length ("hello", [1,2,3])
```

даёт `(5,3)`.

## Система F в Haskell'е

- В Haskell'е система F включается прагмой `RankNTypes`.
- Пример:

```haskell
applyToTuple = (\f -> \(x, y) -> (f x, f y))
  :: (forall a. [a] -> b) -> ([c], [d]) -> (b,b)
applyToTuple length ("hello", [1,2,3])
```

  даёт `(5,3)`.
- `length` имеет полиморфный тип — даже более общий, чем `forall a. [a] -> Int`.

# Система F в Haskell'е

- В Haskell'е система F включается прагмой `RankNTypes`.
- Пример:

```
applyToTuple = (\f -> \(x, y) -> (f x, f y))
  :: (forall a. [a] -> b) -> ([c], [d]) -> (b,b)
applyToTuple length ("hello", [1,2,3])
```

  даёт `(5,3)`.
- `length` имеет полиморфный тип — даже более общий, чем `forall a. [a] -> Int`.
- Выведение типов (в $\lambda_{\to}$) даёт

```
applyToTuple :: (t -> b) -> (t, t) -> (b, b)
```

  — применить к `("hello", [1,2,3])` не получится.

## Типизация в системе F

- Вернёмся к примеру $\lambda x.(xx)$.

## Типизация в системе F

- Вернёмся к примеру $\lambda x.(xx)$.
- Кроме типизации $\big(\forall r.(r \to r)\big) \to \forall r.(r \to r)$, корректной является также типизация $\big(\forall r.((r \to r) \to (r \to r))\big) \to \forall r.((r \to r) \to (r \to r))$.

## Типизация в системе F

- Вернёмся к примеру $\lambda x.(xx)$.
- Кроме типизации $\big(\forall r.(r \to r)\big) \to \forall r.(r \to r)$, корректной является также типизация $\big(\forall r.((r \to r) \to (r \to r))\big) \to \forall r.((r \to r) \to (r \to r))$.
- Оказывается, эти две типизации *несравнимы*, хотя вторая кажется конкретизацией первой.

## Типизация в системе F

- Вернёмся к примеру $\lambda x.(xx)$.
- Кроме типизации $\big(\forall r.(r \to r)\big) \to \forall r.(r \to r)$, корректной является также типизация $\big(\forall r.((r \to r) \to (r \to r))\big) \to \forall r.((r \to r) \to (r \to r))$.
- Оказывается, эти две типизации *несравнимы*, хотя вторая кажется конкретизацией первой.
    - Пусть $A = \forall r.(r \to r)$ и $B = \forall r.((r \to r) \to (r \to r))$.

## Типизация в системе F

- Вернёмся к примеру $\lambda x.(xx)$.
- Кроме типизации $\big(\forall r.(r \to r)\big) \to \forall r.(r \to r)$, корректной является также типизация $\big(\forall r.((r \to r) \to (r \to r))\big) \to \forall r.((r \to r) \to (r \to r))$.
- Оказывается, эти две типизации *несравнимы,* хотя вторая кажется конкретизацией первой.
    - Пусть $A = \forall r.(r \to r)$ и $B = \forall r.((r \to r) \to (r \to r))$.
    - Тип $A$ более абстрактный, чем $B$, значит, ему удовлетворяет «меньше» объектов.

# Типизация в системе F

- Вернёмся к примеру $\lambda x.(xx)$.
- Кроме типизации $\big(\forall r.(r \to r)\big) \to \forall r.(r \to r)$, корректной является также типизация $\big(\forall r.((r \to r) \to (r \to r))\big) \to \forall r.((r \to r) \to (r \to r))$.
- Оказывается, эти две типизации *несравнимы,* хотя вторая кажется конкретизацией первой.
    - Пусть $A = \forall r.(r \to r)$ и $B = \forall r.((r \to r) \to (r \to r))$.
    - Тип $A$ более абстрактный, чем $B$, значит, ему удовлетворяет «меньше» объектов.
    - С другой стороны, у функции типа $A \to A$ «меньше» разнообразие аргументов (это должен быть полиморфный объект типа $\forall r.(r \to r)$), значит, найти такую функцию «проще».

# Типизация в системе F

- Конкретные примеры (в Haskell'е): терм `\f g x -> f g x` имеет тип $B \to B$, но не $A \to A$, а `\f -> (\x z -> z) (f 0)` — наоборот.

# Типизация в системе F

- Конкретные примеры (в Haskell'е): терм `\f g x -> f g x` имеет тип $B \to B$, но не $A \to A$, а `\f -> (\x z -> z) (f 0)` — наоборот.
- Итак, в системе F наиболее общий тип может попросту не существовать, а значит, задача выведения типов некорректна.

# Типизация в системе F

- Конкретные примеры (в Haskell'е): терм `\f g x -> f g x` имеет тип $B \to B$, но не $A \to A$, а `\f -> (\x z -> z) (f 0)` — наоборот.
- Итак, в системе F наиболее общий тип может попросту не существовать, а значит, задача выведения типов некорректна.
- Более того, даже задача *типизуемости* (существования хотя бы одного корректного типа) в системе F алгоритмически неразрешима [J.B. Wells].

# Упражнение

```haskell
{-# LANGUAGE RankNTypes #-}

lxx :: (forall a. a -> a) -> (forall a. a -> a)
lxx = \x -> (x x)

lxx2 :: (forall b. (b->b) -> (b->b)) -> (forall b. (b->b) -> (b->b))
lxx2 = \x -> (x x)

dup = \f -> \x -> f (f x)
```

- Чему равно значение `lxx2 dup (+2) 3` ?
- Корректно ли `lxx dup` ?

# Система Хиндли – Милнера

- Некоторым компромиссом между $\lambda_{\to}$ и $\lambda 2$ является *система типов Хиндли – Милнера.*

# Система Хиндли – Милнера

- Некоторым компромиссом между $\lambda_{\to}$ и $\lambda 2$ является *система типов Хиндли – Милнера.*
- В этой системе кванторы $\forall$ допускаются только *на внешнем уровне,* т.е. типы имеют вид $\forall r_1. \ldots \forall r_k.A$, где $A$ — бескванторный тип.

# Система Хиндли – Милнера

- Некоторым компромиссом между $\lambda_{\to}$ и $\lambda 2$ является *система типов Хиндли – Милнера.*
- В этой системе кванторы $\forall$ допускаются только *на внешнем уровне,* т.е. типы имеют вид $\forall r_1. \dots \forall r_k.A$, где $A$ — бескванторный тип.
- Такие типы могут встречаться у переменных в контекстах, однако $\lambda$-абстракция таких переменных запрещена.

## Система Хиндли – Милнера

- Некоторым компромиссом между $\lambda_{\to}$ и $\lambda 2$ является *система типов Хиндли – Милнера.*
- В этой системе кванторы $\forall$ допускаются только *на внешнем уровне,* т.е. типы имеют вид $\forall r_1. \ldots \forall r_k.A$, где $A$ — бескванторный тип.
- Такие типы могут встречаться у переменных в контекстах, однако $\lambda$-абстракция таких переменных запрещена.
- Тем не менее, имеется дополнительная конструкция термов $\mathbf{let}\ \ldots \mathbf{in}\ \ldots$, которая допускает аналог абстракции по такой полиморфной переменной *(let-полиморфизм).*

# Система Хиндли – Милнера

- Некоторым компромиссом между $\lambda_{\to}$ и $\lambda 2$ является *система типов Хиндли – Милнера.*
- В этой системе кванторы $\forall$ допускаются только *на внешнем уровне,* т.е. типы имеют вид $\forall r_1. \ldots \forall r_k.A$, где $A$ — бескванторный тип.
- Такие типы могут встречаться у переменных в контекстах, однако $\lambda$-абстракция таких переменных запрещена.
- Тем не менее, имеется дополнительная конструкция термов **let** … **in** …, которая допускает аналог абстракции по такой полиморфной переменной *(let-полиморфизм).*
- Кроме того, в конструкцию **let** … **in** … встроена рекурсия.

# Система Хиндли – Милнера

- Некоторым компромиссом между $\lambda_{\to}$ и $\lambda 2$ является *система типов Хиндли – Милнера.*
- В этой системе кванторы $\forall$ допускаются только *на внешнем уровне,* т.е. типы имеют вид $\forall r_1. \dots \forall r_k.A$, где $A$ — бескванторный тип.
- Такие типы могут встречаться у переменных в контекстах, однако $\lambda$-абстракция таких переменных запрещена.
- Тем не менее, имеется дополнительная конструкция термов **let** … **in** …, которая допускает аналог абстракции по такой полиморфной переменной *(let-полиморфизм).*
- Кроме того, в конструкцию **let** … **in** … встроена рекурсия.
- И главное: система Хиндли – Милнера допускает выведение наиболее общего типа!

# Let-полиморфизм

- Возможность полиморфного типа в контексте уже расширяет допустимые типизации, например: $x : \forall r.(r \to r) \vdash (xx) : \forall s.(s \to s)$. (Здесь у первого $x$’а тип $(s \to s) \to (s \to s)$, у второго — $(s \to s)$.)

# Let-полиморфизм

- Возможность полиморфного типа в контексте уже расширяет допустимые типизации, например: $x : \forall r.(r \to r) \vdash (xx) : \forall s.(s \to s)$. (Здесь у первого $x$'а тип $(s \to s) \to (s \to s)$, у второго — $(s \to s)$.)
- Однако более интересные вещи связаны с новым конструктором термов, **let** ... **in** ....

# Let-полиморфизм

- Возможность полиморфного типа в контексте уже расширяет допустимые типизации, например: $x : \forall r.(r \to r) \vdash (xx) : \forall s.(s \to s)$. (Здесь у первого $x$'а тип $(s \to s) \to (s \to s)$, у второго — $(s \to s)$.)
- Однако более интересные вещи связаны с новым конструктором термов, **let** … **in** ….
- Термы с помощью этого конструктора строятся так: **let** $x = v$ **in** $u$.

## Let-полиморфизм

- Возможность полиморфного типа в контексте уже расширяет допустимые типизации, например: $x : \forall r.(r \to r) \vdash (xx) : \forall s.(s \to s)$. (Здесь у первого $x$’а тип $(s \to s) \to (s \to s)$, у второго — $(s \to s)$.)
- Однако более интересные вещи связаны с новым конструктором термов, **let** … **in** ….
- Термы с помощью этого конструктора строятся так: **let** $x = v$ **in** $u$.
- Пока будем считать, что $x$ не входит свободно в $u$.

# Let-полиморфизм

- Возможность полиморфного типа в контексте уже расширяет допустимые типизации, например: $x : \forall r.(r \to r) \vdash (xx) : \forall s.(s \to s)$. (Здесь у первого $x$'а тип $(s \to s) \to (s \to s)$, у второго — $(s \to s)$.)
- Однако более интересные вещи связаны с новым конструктором термов, **let** … **in** ….
- Термы с помощью этого конструктора строятся так: $\mathbf{let}\ x = v\ \mathbf{in}\ u$.
- Пока будем считать, что $x$ не входит свободно в $u$.
- Редукция: $(\mathbf{let}\ x = v\ \mathbf{in}\ u) \to u[x := v]$.

# Let-полиморфизм

- Возможность полиморфного типа в контексте уже расширяет допустимые типизации, например: $x : \forall r.(r \to r) \vdash (xx) : \forall s.(s \to s)$. (Здесь у первого $x$'а тип $(s \to s) \to (s \to s)$, у второго — $(s \to s)$.)
- Однако более интересные вещи связаны с новым конструктором термов, **let** ... **in** ....
- Термы с помощью этого конструктора строятся так: $\mathbf{let}\ x = v\ \mathbf{in}\ u$.
- Пока будем считать, что $x$ не входит свободно в $u$.
- Редукция: $\big(\mathbf{let}\ x = v\ \mathbf{in}\ u\big) \to u[x := v]$.
    - Таким образом, при бестиповом подходе $\mathbf{let}\ x = v\ \mathbf{in}\ u$ — это то же, что и $(\lambda x.u)v$.

# Let-полиморфизм

- С точки зрения типов, однако, $\mathbf{let}\ x = v\ \mathbf{in}\ u$ отличается от $(\lambda x.u)v$, поскольку в первом случае переменная $x$ *может иметь полиморфный тип:*

$$\frac{\Gamma \vdash v : A \quad \Gamma, x : A \vdash u : B}{\Gamma \vdash \left(\mathbf{let}\ x = v\ \mathbf{in}\ u\right) : B}\ \text{Let}$$

# Let-полиморфизм

- С точки зрения типов, однако, $\textbf{let}\ x = v\ \textbf{in}\ u$ отличается от $(\lambda x.u)v$, поскольку в первом случае переменная $x$ *может иметь полиморфный тип:*

$$\frac{\Gamma \vdash v : A \quad \Gamma, x : A \vdash u : B}{\Gamma \vdash \big(\textbf{let}\ x = v\ \textbf{in}\ u\big) : B}\ \text{Let}$$

- Для удобства примеров будем считать, что у нас есть конструкция пары:

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B}{\Gamma \vdash (u, v) : (A, B)}\ \text{Pair}$$

$$\textbf{fst} : \forall rs.((r, s) \to r) \qquad \textbf{snd} : \forall rs.((r, s) \to s)$$

и какие-нибудь базовые типы (например, `Int` и `Char`).

- Тогда мы можем типизовать $\mathbf{let}\ f = \lambda x.x\ \mathbf{in}\ (f\ 0, f\ \text{‘a’})$ как $(\mathtt{Int}, \mathtt{Char})$, а вот $\big(\lambda f.(f\ 0, f\ \text{‘a’})\big)\,(\lambda x.x)$ в системе Хиндли – Милнера (в отличие от системы F) нетипизуем.
- В системе Хиндли – Милнера у любого типизуемого (в данном контексте) терма есть наиболее общий тип!
- Задача поиска наиболее общего типа алгоритмически разрешима.
- Основная идея: когда мы выводим тип $\mathbf{let}\ v = x\ \mathbf{in}\ u$, сначала находим наиболее общий тип $A$ для $v$, потом подставляем его в качестве типа для $x$, и в контексте $\Gamma, x : A$ типизуем $u$.

# Система типов Хиндли – Милнера

- При этом сама конструкция **let** может быть под лямбдами, поэтому тип $A$ в дальнейшем выводе может быть уточнён (конкретизирован).

## Система типов Хиндли – Милнера

- При этом сама конструкция **let** может быть под лямбдами, поэтому тип $A$ в дальнейшем выводе может быть уточнён (конкретизирован).
- Например, терм $u = \lambda z.\big(\mathbf{let}\ f = \lambda x.z\ \mathbf{in}\ f\ 0\big)$ имеет наиболее общий тип $r \to r$ (точнее, $\forall r.(r \to r)$), а вот $(u\ \text{`a'})$ — уже более конкретный тип `Char`.

# Система типов Хиндли – Милнера

- При этом сама конструкция **let** может быть под лямбдами, поэтому тип $A$ в дальнейшем выводе может быть уточнён (конкретизирован).
- Например, терм $u = \lambda z.\big(\mathbf{let}\ f = \lambda x.z\ \mathbf{in}\ f\,0\big)$ имеет наиболее общий тип $r \to r$ (точнее, $\forall r.(r \to r)$), а вот $(u\,\text{‘a’})$ — уже более конкретный тип $\mathtt{Char}$.
    - Терм $(u\,(\lambda y.y))$ имеет также полиморфный тип $\forall s.(s \to s)$.

# Система типов Хиндли – Милнера

- При этом сама конструкция **let** может быть под лямбдами, поэтому тип $A$ в дальнейшем выводе может быть уточнён (конкретизирован).
- Например, терм $u = \lambda z.\big(\textbf{let } f = \lambda x.z \textbf{ in } f\,0\big)$ имеет наиболее общий тип $r \to r$ (точнее, $\forall r.(r \to r)$), а вот $(u\,\text{‘a’})$ — уже более конкретный тип $\texttt{Char}$.
    - Терм $(u\,(\lambda y.y))$ имеет также полиморфный тип $\forall s.(s \to s)$.
    - Конкретизация происходит на последнем шаге вывода:

$$\frac{\vdash \lambda z.(\textbf{let } f = \lambda x.z \textbf{ in } f\,0) : r_1 \to r_1 \quad \vdash \text{‘a’} : \texttt{Char}}{\vdash \big(\lambda z.(\textbf{let } f = \lambda x.z \textbf{ in } f\,0)\big)\,\text{‘a’} : r_0}\ \text{App}$$

при унификации $(r_1 \to r_1) \approx (\texttt{Char} \to r_0)$
(даёт подстановку $r_1 := \texttt{Char}, r_0 := r_1$).

## Система типов Хиндли – Милнера

- Чтобы аккуратно разобраться с выведением типов в системе Хиндли – Милнера, сначала выпишем её правила типизации, а потом изложим алгоритм J для выведения типов.

# Система типов Хиндли – Милнера

- Чтобы аккуратно разобраться с выведением типов в системе Хиндли – Милнера, сначала выпишем её правила типизации, а потом изложим алгоритм J для выведения типов.
- Исчисление для типизации будет просто фрагментом исчисления $\lambda 2$ (система F), с ограничением на использование квантора $\forall$ и добавленным конструктором **let**.

# Система типов Хиндли – Милнера

- Чтобы аккуратно разобраться с выведением типов в системе Хиндли – Милнера, сначала выпишем её правила типизации, а потом изложим алгоритм J для выведения типов.
- Исчисление для типизации будет просто фрагментом исчисления $\lambda 2$ (система F), с ограничением на использование квантора $\forall$ и добавленным конструктором **let**.
- Алгоритм J напоминает алгоритм с «желательными равенствами» для $\lambda_{\rightarrow}$, однако унификация будет осуществляться не один раз в конце, а «по ходу дела».

# Система типов Хиндли – Милнера

*Декларативная* система правил:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{Ax} \qquad \frac{\Gamma \vdash u : (A \to B) \quad \Gamma \vdash v : A}{\Gamma \vdash (uv) : B}\ \text{App}$$

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash (\lambda x.u) : (A \to B)}\ \text{Abs}; \quad A \text{ бескванторный}$$

$$\frac{\Gamma \vdash v : A \quad \Gamma, x : A \vdash u : B}{\Gamma \vdash (\mathbf{let}\ x = v\ \mathbf{in}\ u) : B}\ \text{Let}$$

$$\frac{\Gamma \vdash u : A}{\Gamma \vdash u : (\forall r.A)}\ \text{Gen}; \ r \notin \text{FreeVar}(\Gamma)$$

$$\frac{\Gamma \vdash u : (\forall r.B)}{\Gamma \vdash u : B[r := A]}\ \text{Inst}; \ \text{при условии корректности подстановки}$$

# Система типов Хиндли – Милнера

*Декларативная* система правил:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{Ax} \qquad \frac{\Gamma \vdash u : (A \to B) \quad \Gamma \vdash v : A}{\Gamma \vdash (uv) : B}\ \text{App}$$

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash (\lambda x.u) : (A \to B)}\ \text{Abs};\quad A\ \text{бескванторный}$$

$$\frac{\Gamma \vdash v : A \quad \Gamma, x : A \vdash u : B}{\Gamma \vdash (\textbf{let}\ x = v\ \textbf{in}\ u) : B}\ \text{Let} \qquad (\text{без ограничений на } A)$$

$$\frac{\Gamma \vdash u : A}{\Gamma \vdash u : (\forall r.A)}\ \text{Gen};\ r \notin \text{FreeVar}(\Gamma)$$

$$\frac{\Gamma \vdash u : (\forall r.B)}{\Gamma \vdash u : B[r := A]}\ \text{Inst};\ \text{при условии корректности подстановки}$$

# Алгоритм J

- *Структура* дерева вывода для $\Gamma \vdash u : \mathbf{?}$ однозначно определяется термом $u$.

# Алгоритм J

- *Структура* дерева вывода для $\Gamma \vdash u : \textbf{?}$ однозначно определяется термом $u$.
- Алгоритм J расставляет в этом дереве типы.

# Алгоритм J

- *Структура* дерева вывода для $\Gamma \vdash u : \mathbf{?}$ однозначно определяется термом $u$.
- Алгоритм J расставляет в этом дереве типы.
- Алгоритм обходит дерево рекурсивно, проходя посылки каждого правила *слева направо.*

# Алгоритм J

- *Структура* дерева вывода для $\Gamma \vdash u : ?$ однозначно определяется термом $u$.
- Алгоритм J расставляет в этом дереве типы.
- Алгоритм обходит дерево рекурсивно, проходя посылки каждого правила *слева направо.*
- Кроме посылок, у правил также будут «императивные» команды, меняющие уже назначенные типы.

# Алгоритм J

- *Структура* дерева вывода для $\Gamma \vdash u : \textbf{?}$ однозначно определяется термом $u$.
- Алгоритм J расставляет в этом дереве типы.
- Алгоритм обходит дерево рекурсивно, проходя посылки каждого правила *слева направо.*
- Кроме посылок, у правил также будут «императивные» команды, меняющие уже назначенные типы.
- Для этого поддерживается «глобальная» подстановка, к которой добавляются уточняющие подстановки посредством композиции.

# Алгоритм J

- *Структура* дерева вывода для $\Gamma \vdash u : \mathbf{?}$ однозначно определяется термом $u$.
- Алгоритм J расставляет в этом дереве типы.
- Алгоритм обходит дерево рекурсивно, проходя посылки каждого правила *слева направо.*
- Кроме посылок, у правил также будут «императивные» команды, меняющие уже назначенные типы.
- Для этого поддерживается «глобальная» подстановка, к которой добавляются уточняющие подстановки посредством композиции.
- «Чистая» реализация алгоритма J, где подстановки везде указаны явно, называется алгоритмом W.

## Алгоритм J

- Императивные команды:

## Алгоритм J

- Императивные команды:
  - $r = \mathsf{newvar}$ — ввести новую переменную $r$.

# Алгоритм J

- Императивные команды:
  - $r = \mathsf{newvar}$ — ввести новую переменную $r$.
  - $A' = \mathsf{inst}(A)$ — если $A = \forall r_1 \dots r_k.B$, где $B$ бескванторный, то $A' = B[r_1 := r_1', \dots, r_k := r_k']$, где $r_i'$ новые.

# Алгоритм J

- Императивные команды:
  - $r = \mathsf{newvar}$ — ввести новую переменную $r$.
  - $A' = \mathsf{inst}(A)$ — если $A = \forall r_1 \dots r_k.B$, где $B$ бескванторный, то $A' = B[r_1 := r_1', \dots, r_k := r_k']$, где $r_i'$ новые.
  - $\mathsf{unify}(A \approx B)$ — найти MGU типов $A$ и $B$ и уточнить им глобальную подстановку.

# Алгоритм J

- Императивные команды:
    - $r = \mathsf{newvar}$ — ввести новую переменную $r$.
    - $A' = \mathsf{inst}(A)$ — если $A = \forall r_1 \dots r_k.B$, где $B$ бескванторный, то $A' = B[r_1 := r_1', \dots, r_k := r_k']$, где $r_i'$ новые.
    - $\mathsf{unify}(A \approx B)$ — найти MGU типов $A$ и $B$ и уточнить им глобальную подстановку.
- Через $\forall\bar{\Gamma}.A$ обозначим замыкание типа $A$ кванторами $\forall$ по всем переменным, не входящим в $\Gamma$.

# Алгоритм J

- Императивные команды:
  - $r = \mathsf{newvar}$ — ввести новую переменную $r$.
  - $A' = \mathsf{inst}(A)$ — если $A = \forall r_1 \dots r_k.B$, где $B$ бескванторный, то $A' = B[r_1 := r_1', \dots, r_k := r_k']$, где $r_i'$ новые.
  - $\mathsf{unify}(A \approx B)$ — найти MGU типов $A$ и $B$ и уточнить им глобальную подстановку.
- Через $\forall\bar{\Gamma}.A$ обозначим замыкание типа $A$ кванторами $\forall$ по всем переменным, не входящим в $\Gamma$.
- Договоримся, что кванторы будут использоваться только в левой части (контекст). В правой части будут свежие свободные переменные.

# Алгоритм J

- Императивные команды:
  - $r = \mathsf{newvar}$ — ввести новую переменную $r$.
  - $A' = \mathsf{inst}(A)$ — если $A = \forall r_1 \dots r_k . B$, где $B$ бескванторный, то $A' = B[r_1 := r'_1, \dots, r_k := r'_k]$, где $r'_i$ новые.
  - $\mathsf{unify}(A \approx B)$ — найти MGU типов $A$ и $B$ и уточнить им глобальную подстановку.
- Через $\forall\bar{\Gamma}.A$ обозначим замыкание типа $A$ кванторами $\forall$ по всем переменным, не входящим в $\Gamma$.
- Договоримся, что кванторы будут использоваться только в левой части (контекст). В правой части будут свежие свободные переменные.
  - Это избавляет от правил Gen и Inst.

## Алгоритм J: «исчисление»

$$\frac{(x : A) \in \Gamma \qquad A' = \mathsf{inst}(A)}{\Gamma \vdash_J x : A'}\ \text{AxInst}$$

$$\frac{\Gamma \vdash_J u : E \quad \Gamma \vdash_J v : F \quad r = \mathsf{newvar} \quad \mathsf{unify}(E \approx F \to r)}{\Gamma \vdash_J (uv) : r}\ \text{App}$$

$$\frac{r = \mathsf{newvar} \quad \Gamma, x : r \vdash_J u : B}{\Gamma \vdash_J (\lambda x.u) : (r \to B)}\ \text{Abs}$$

$$\frac{\Gamma \vdash_J v : A \quad \Gamma, x : \forall\bar{\Gamma}.A \vdash_J u : B}{\Gamma \vdash_J (\mathbf{let}\ x = v\ \mathbf{in}\ u) : B}\ \text{Let}$$

# Алгоритм J

- Понятие наиболее общего типа определяется так же, как и для $\lambda_{\rightarrow}$.

# Алгоритм J

- Понятие наиболее общего типа определяется так же, как и для $\lambda_{\to}$.
    - Кванторов в правой части нет, поэтому трудностей с подстановками не возникает.

# Алгоритм J

- Понятие наиболее общего типа определяется так же, как и для $\lambda_{\to}$.
    - Кванторов в правой части нет, поэтому трудностей с подстановками не возникает.
- Можно доказать **теорему** о том, что алгоритм J находит этот самый наиболее общий тип — или выясняет, что его нет, в таком случае где-то не срабатывает unify.

# Алгоритм J

- Понятие наиболее общего типа определяется так же, как и для $\lambda_{\to}$.
  - Кванторов в правой части нет, поэтому трудностей с подстановками не возникает.
- Можно доказать **теорему** о том, что алгоритм J находит этот самый наиболее общий тип — или выясняет, что его нет, в таком случае где-то не срабатывает unify.
  - Для доказательства удобнее использовать алгоритм W.

# Алгоритм J: пример

- Типизируем терм **let** $z = \lambda f.\lambda x.f(fx)$ **in** $(zz)$.

## Алгоритм J: пример

- Типизируем терм **let** $z = \lambda f.\lambda x.f(fx)$ **in** $(zz)$.
- К сожалению, всё дерево не влезет на слайд, поэтому будем отображать его по частям.

## Алгоритм J: пример

- Типизируем терм **let** $z = \lambda f.\lambda x.f(fx)$ **in** $(zz)$.
- К сожалению, всё дерево не влезет на слайд, поэтому будем отображать его по частям.

$$\dfrac{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : A \qquad\qquad z : \forall\bar{\Gamma}.A \vdash_J (zz) : \mathbf{??}}{\vdash_J \big(\mathbf{let}\ z = \lambda f.\lambda x.f(fx)\ \mathbf{in}\ (zz)\big) : \mathbf{?}}\ \text{Let}$$

# Алгоритм J: пример

- Типизируем терм **let** $z = \lambda f.\lambda x.f(fx)$ **in** $(zz)$.
- К сожалению, всё дерево не влезет на слайд, поэтому будем отображать его по частям.

$$\dfrac{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : A \qquad\qquad z : \forall\bar{\Gamma}.A \vdash_J (zz) : \mathbf{??}}{\vdash_J \big(\mathbf{let}\ z = \lambda f.\lambda x.f(fx)\ \mathbf{in}\ (zz)\big) : \mathbf{?}}\ \text{Let}$$

- Для замкнутого терма (комбинатора), не содержащего **let**, алгоритм J работает так же, как и выведение типов в $\lambda_{\to}$.

## Алгоритм J: пример

- Типизируем терм **let** $z = \lambda f.\lambda x.f(fx)$ **in** $(zz)$.
- К сожалению, всё дерево не влезет на слайд, поэтому будем отображать его по частям.

$$\dfrac{\vdash_J \left(\lambda f.\lambda x.f(fx)\right) : (q \to q) \to (q \to q) \qquad\qquad z : \forall\bar{\Gamma}.A \vdash_J (zz) : \mathbf{??}}{\vdash_J \left(\mathbf{let}\ z = \lambda f.\lambda x.f(fx)\ \mathbf{in}\ (zz)\right) : \mathbf{?}}\ \text{Let}$$

- Для замкнутого терма (комбинатора), не содержащего **let**, алгоритм J работает так же, как и выведение типов в $\lambda_{\to}$.

## Алгоритм J: пример

- Типизируем терм $\textbf{let}\ z = \lambda f.\lambda x.f(fx)\ \textbf{in}\ (zz)$.
- К сожалению, всё дерево не влезет на слайд, поэтому будем отображать его по частям.

$$\dfrac{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : (q \to q) \to (q \to q) \qquad z : \forall\bar{\Gamma}.A \vdash_J (zz) : \textbf{??}}{\vdash_J \big(\textbf{let}\ z = \lambda f.\lambda x.f(fx)\ \textbf{in}\ (zz)\big) : \textbf{?}}\ \text{Let}$$

- Для замкнутого терма (комбинатора), не содержащего **let**, алгоритм J работает так же, как и выведение типов в $\lambda_{\to}$.
    - Потом мы посмотрим на это подробнее.

# Алгоритм J: пример

- Типизируем терм **let** $z = \lambda f.\lambda x.f(fx)$ **in** $(zz)$.
- К сожалению, всё дерево не влезет на слайд, поэтому будем отображать его по частям.

$$\dfrac{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : (q \to q) \to (q \to q) \qquad\qquad z : \forall\bar{\Gamma}.A \vdash_J (zz) : \mathbf{??}}{\vdash_J \big(\mathbf{let}\ z = \lambda f.\lambda x.f(fx)\ \mathbf{in}\ (zz)\big) : \mathbf{?}}\ \text{Let}$$

- Для замкнутого терма (комбинатора), не содержащего **let**, алгоритм J работает так же, как и выведение типов в $\lambda_\to$.
  - Потом мы посмотрим на это подробнее.
- Теперь рекурсивно применяем алгоритм к $z : \forall q.((q \to q) \to (q \to q)) \vdash (zz) : \mathbf{??}$.

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\dfrac{\Gamma_z \vdash_J z : (s \to s) \to (s \to s)}{\Gamma_z \vdash_J z : (t \to t) \to (t \to t)} \quad \dfrac{r = \mathsf{newvar}}{\mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big)}}{\Gamma_z \vdash_J (zz) : r}\ \text{App}$$

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\dfrac{}{\Gamma_z \vdash_J z : (s \to s) \to (s \to s)} \quad \dfrac{}{\Gamma_z \vdash_J z : (t \to t) \to (t \to t)} \quad \dfrac{r = \mathsf{newvar}}{\mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big)}}{\Gamma_z \vdash_J (zz) : r}\ \text{App}$$

- Две посылки слева получены применением AxInst ($s$ и $t$ — новые переменные).

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\begin{array}{ll} \Gamma_z \vdash_J z : (s \to s) \to (s \to s) & r = \mathsf{newvar} \\ \Gamma_z \vdash_J z : (t \to t) \to (t \to t) & \mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big) \end{array}}{\Gamma_z \vdash_J (zz) : r}\ \text{App}$$

- Две посылки слева получены применением AxInst ($s$ и $t$ — новые переменные).
- Унифицируем:

$$s \to s \approx (t \to t) \to (t \to t)$$
$$s \to s \approx r$$

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\begin{array}{ll} \Gamma_z \vdash_J z : (s \to s) \to (s \to s) & r = \mathsf{newvar} \\ \Gamma_z \vdash_J z : (t \to t) \to (t \to t) & \mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big) \end{array}}{\Gamma_z \vdash_J (zz) : r}\ \text{App}$$

- Две посылки слева получены применением AxInst ($s$ и $t$ — новые переменные).
- Унифицируем:

$$s \to s \approx (t \to t) \to (t \to t)$$
$$r := s \to s$$

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\begin{array}{cc} \Gamma_z \vdash_J z : (s \to s) \to (s \to s) & r = \mathsf{newvar} \\ \Gamma_z \vdash_J z : (t \to t) \to (t \to t) & \mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big) \end{array}}{\Gamma_z \vdash_J (zz) : r}\ \text{App}$$

- Две посылки слева получены применением AxInst ($s$ и $t$ — новые переменные).
- Унифицируем:

$$s \approx t \to t$$

$$r := s \to s$$

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\begin{array}{ll} \Gamma_z \vdash_J z : (s \to s) \to (s \to s) & r = \mathsf{newvar} \\ \Gamma_z \vdash_J z : (t \to t) \to (t \to t) & \mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big) \end{array}}{\Gamma_z \vdash_J (zz) : r}\ \text{App}$$

- Две посылки слева получены применением AxInst ($s$ и $t$ — новые переменные).
- Унифицируем:

$$s := t \to t$$

$$r := s \to s$$

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\begin{array}{cc} \Gamma_z \vdash_J z : (s \to s) \to (s \to s) & r = \mathsf{newvar} \\ \Gamma_z \vdash_J z : (t \to t) \to (t \to t) & \mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big) \end{array}}{\Gamma_z \vdash_J (zz) : r}\ \mathrm{App}$$

- Две посылки слева получены применением AxInst ($s$ и $t$ — новые переменные).
- Унифицируем:

$$s := t \to t$$
$$r := (t \to t) \to (t \to t)$$

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\begin{array}{cc} \Gamma_z \vdash_J z : (s \to s) \to (s \to s) & r = \mathsf{newvar} \\ \Gamma_z \vdash_J z : (t \to t) \to (t \to t) & \mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big) \end{array}}{\Gamma_z \vdash_J (zz) : r}\ \mathrm{App}$$

- Две посылки слева получены применением AxInst ($s$ и $t$ — новые переменные).
- Унифицируем:

$$\begin{aligned} s &:= t \to t \\ r &:= (t \to t) \to (t \to t) \end{aligned}$$

- Применяя Let, окончательно получаем $\vdash_J \big(\textbf{let}\ z = \lambda f.\lambda x.f(fx)\ \textbf{in}\ (zz)\big) : (t \to t) \to (t \to t)$.

# Алгоритм J: пример

Пусть $\Gamma_z = z : \forall q.((q \to q) \to (q \to q))$

$$\dfrac{\begin{array}{ll} \Gamma_z \vdash_J z : (s \to s) \to (s \to s) & r = \mathsf{newvar} \\ \Gamma_z \vdash_J z : (t \to t) \to (t \to t) & \mathsf{unify}\big((s \to s) \to (s \to s) \approx ((t \to t) \to (t \to t)) \to r\big) \end{array}}{\Gamma_z \vdash_J (zz) : r}\ \text{App}$$

- Две посылки слева получены применением AxInst ($s$ и $t$ — новые переменные).
- Унифицируем:

$$s := t \to t$$
$$r := (t \to t) \to (t \to t)$$

- Применяя Let, окончательно получаем $\vdash_J \big(\mathbf{let}\ z = \lambda f.\lambda x.f(fx)\ \mathbf{in}\ (zz)\big) : (t \to t) \to (t \to t)$.
- Сверху можно «навесить» $\forall t$.

# Алгоритм J: пример

Вернёмся к $\lambda f.\lambda x.f(fx)$.

$$
\dfrac{
f : r \vdash_J f : r
\qquad
\dfrac{f : r \vdash_J f : r \quad x : q \vdash_J x : q \quad \mathsf{unify}(r \approx q \to s)}{f : r, x : q \vdash_J (fx) : s}\ \text{App}
\qquad
\mathsf{unify}(r \approx s \to p)
}{
\dfrac{
\dfrac{f : r, x : q \vdash_J f(fx) : p}{f : r \vdash_J \lambda x.f(fx) : q \to p}\ \text{Abs}
}{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : r \to (q \to p)}\ \text{Abs}
}\ \text{App}
$$

# Алгоритм J: пример

Вернёмся к $\lambda f.\lambda x.f(fx)$.

$$\dfrac{\dfrac{f : r \vdash_J f : r \qquad \dfrac{f : r \vdash_J f : r \quad x : q \vdash_J x : q \quad \color{blue}{r := q \to s}}{f : r, x : q \vdash_J (fx) : s}\,\text{App} \qquad \text{unify}(r \approx s \to p)}{\dfrac{\dfrac{f : r, x : q \vdash_J f(fx) : p}{f : r \vdash_J \lambda x.f(fx) : q \to p}\,\text{Abs}}{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : r \to (q \to p)}\,\text{Abs}}\,\text{App}$$

Вернёмся к $\lambda f.\lambda x.f(fx)$.

$$\dfrac{\dfrac{f : q \to s \vdash_J f : q \to s \qquad \dfrac{f : q \to s \vdash_J f : q \to s \quad x : q \vdash_J x : q}{f : q \to s, x : q \vdash_J (fx) : s}\;\text{App} \qquad \text{unify}(q \to s \approx s \to p)}{f : q \to s, x : q \vdash_J f(fx) : p}\;\text{App}}{\dfrac{f : q \to s \vdash_J \lambda x.f(fx) : q \to p}{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : (q \to s) \to (q \to p)}\;\text{Abs}}\;\text{Abs}$$

# Алгоритм J: пример

Вернёмся к $\lambda f.\lambda x.f(fx)$.

$$\dfrac{\dfrac{f : q \to s \vdash_J f : q \to s \qquad \dfrac{f : q \to s \vdash_J f : q \to s \quad x : q \vdash_J x : q}{f : q \to s, x : q \vdash_J (fx) : s}\text{ App} \qquad \text{unify}(q \approx s, s \approx p)}{\dfrac{f : q \to s, x : q \vdash_J f(fx) : p}{f : q \to s \vdash_J \lambda x.f(fx) : q \to p}\text{ Abs}}\text{ App}}{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : (q \to s) \to (q \to p)}\text{ Abs}$$

# Алгоритм J: пример

Вернёмся к $\lambda f.\lambda x.f(fx)$.

$$\dfrac{\dfrac{f : q \to s \vdash_J f : q \to s \qquad \dfrac{f : q \to s \vdash_J f : q \to s \quad x : q \vdash_J x : q}{f : q \to s, x : q \vdash_J (fx) : s}\ \text{App} \qquad \color{blue}{s := q, p := q}}{\dfrac{f : q \to s, x : q \vdash_J f(fx) : p}{f : q \to s \vdash_J \lambda x.f(fx) : q \to p}\ \text{Abs}}\ \text{App}}{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : (q \to s) \to (q \to p)}\ \text{Abs}$$

# Алгоритм J: пример

Вернёмся к $\lambda f.\lambda x.f(fx)$.

$$\dfrac{\dfrac{f : q \to q \vdash_J f : q \to q \qquad \dfrac{f : q \to q \vdash_J f : q \to q \quad x : q \vdash_J x : q}{f : q \to q, x : q \vdash_J (fx) : q}\ \text{App}}{\dfrac{f : q \to q, x : q \vdash_J f(fx) : q}{f : q \to q \vdash_J \lambda x.f(fx) : q \to q}\ \text{Abs}}\ \text{App}}{\vdash_J \big(\lambda f.\lambda x.f(fx)\big) : (q \to q) \to (q \to q)}\ \text{Abs}$$

# Рекурсивный let

- **let** бывает рекурсивным.

# Рекурсивный let

- **let** бывает рекурсивным.
- В нашем определении мы запрещали переменной $x$ свободно входить в $v$ при построении терм **let** $x = v$ **in** $u$.

# Рекурсивный let

- **let** бывает рекурсивным.
- В нашем определении мы запрещали переменной $x$ свободно входить в $v$ при построении терм $\textbf{let } x = v \textbf{ in } u$.
  - Трудность в том, что $x$ не входит в $\Gamma$ (это новая переменная).

# Рекурсивный let

- **let** бывает рекурсивным.
- В нашем определении мы запрещали переменной $x$ свободно входить в $v$ при построении терм **let** $x = v$ **in** $u$.
    - Трудность в том, что $x$ не входит в $\Gamma$ (это новая переменная).
- Самый простой способ — использовать операцию взятия неподвижной точки, добавив в контекст константу $\mathbb{Y} : \forall r.((r \to r) \to r)$ и записав рекурсивный **let** как **let** $x = \mathbb{Y}(\lambda x.v)$ **in** $u$.

# Рекурсивный let

- **let** бывает рекурсивным.
- В нашем определении мы запрещали переменной $x$ свободно входить в $v$ при построении терм $\mathbf{let}\ x = v\ \mathbf{in}\ u$.
    - Трудность в том, что $x$ не входит в $\Gamma$ (это новая переменная).
- Самый простой способ — использовать операцию взятия неподвижной точки, добавив в контекст константу $\mathbb{Y} : \forall r.((r \to r) \to r)$ и записав рекурсивный **let** как $\mathbf{let}\ x = \mathbb{Y}(\lambda x.v)\ \mathbf{in}\ u$.
- Несмотря на то, что переменная $x$ здесь оказалась под $\lambda$'ой, let-полиморфизм не пропадает (подумайте почему!).

# Упражнение

Выведите наиболее общий тип с рекурсивным **let**:

$$\textbf{let } y = \lambda f.(f(f(y f))) \textbf{ in } y$$

# Алгебраические типы

- В Haskell’е также имеются *алгебраические типы*, причём они могут иметь параметры.

# Алгебраические типы

- В Haskell'е также имеются *алгебраические типы*, причём они могут иметь параметры.
    - Классический пример — тип списка `[a]`, но мы сначала рассмотрим более простой тип `Maybe a`.

# Алгебраические типы

- В Haskell'е также имеются *алгебраические типы*, причём они могут иметь параметры.
    - Классический пример — тип списка `[a]`, но мы сначала рассмотрим более простой тип `Maybe a`.
    - Этот тип имеет два *конструктора:* `Nothing` и `Just a`.

# Алгебраические типы

- В Haskell'е также имеются *алгебраические типы*, причём они могут иметь параметры.
  - Классический пример — тип списка `[a]`, но мы сначала рассмотрим более простой тип `Maybe a`.
  - Этот тип имеет два *конструктора:* `Nothing` и `Just a`.
  - Всякий объект типа - это либо `Nothing`, либо `Just x`, где `x :: a`.

# Алгебраические типы

- В Haskell'е также имеются *алгебраические типы*, причём они могут иметь параметры.
    - Классический пример — тип списка `[a]`, но мы сначала рассмотрим более простой тип `Maybe a`.
    - Этот тип имеет два *конструктора:* `Nothing` и `Just a`.
    - Всякий объект типа - это либо `Nothing`, либо `Just x`, где `x :: a`.
    - Функции на алгебраическом типе можно определять разбором случаев. Имеются соответствующие редукции.

# Алгебраические типы

- В Haskell'е также имеются *алгебраические типы*, причём они могут иметь параметры.
  - Классический пример — тип списка `[a]`, но мы сначала рассмотрим более простой тип `Maybe a`.
  - Этот тип имеет два *конструктора:* `Nothing` и `Just a`.
  - Всякий объект типа - это либо `Nothing`, либо `Just x`, где `x :: a`.
  - Функции на алгебраическом типе можно определять разбором случаев. Имеются соответствующие редукции.
  - Всё это совместимо с выведением типов по Хиндли – Милнеру.

# Алгебраические типы

- В Haskell'е также имеются *алгебраические типы*, причём они могут иметь параметры.
  - Классический пример — тип списка `[a]`, но мы сначала рассмотрим более простой тип `Maybe a`.
  - Этот тип имеет два *конструктора:* `Nothing` и `Just a`.
  - Всякий объект типа - это либо `Nothing`, либо `Just x`, где `x :: a`.
  - Функции на алгебраическом типе можно определять разбором случаев. Имеются соответствующие редукции.
  - Всё это совместимо с выведением типов по Хиндли – Милнеру.
- Почему такие типы называются алгебраическими?

# Алгебраические типы: произведение

- Построение алгебраического типа сводится к двум базовым операциями — *произведения* и *суммы.*

# Алгебраические типы: произведение

- Построение алгебраического типа сводится к двум базовым операциями — *произведения* и *суммы.*
- Построение объекта с помощью конструктора — это (декартово) произведение.

# Алгебраические типы: произведение

- Построение алгебраического типа сводится к двум базовым операциями — *произведения* и *суммы.*
- Построение объекта с помощью конструктора — это (декартово) произведение.
- Простейший пример (к которому всё сводится) — это уже знакомая нам пара из двух элементов:

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B}{\Gamma \vdash (u, v) : (A, B)}\ \text{Pair}$$

$$\mathbf{fst} : \forall rs.((r, s) \to r) \qquad \mathbf{snd} : \forall rs.((r, s) \to s)$$

# Алгебраические типы: произведение

- Построение алгебраического типа сводится к двум базовым операциями — *произведения* и *суммы.*
- Построение объекта с помощью конструктора — это (декартово) произведение.
- Простейший пример (к которому всё сводится) — это уже знакомая нам пара из двух элементов:

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B}{\Gamma \vdash (u, v) : (A, B)}\ \text{Pair}$$

$$\mathbf{fst} : \forall rs.((r, s) \to r) \qquad \mathbf{snd} : \forall rs.((r, s) \to s)$$

- Если у конструктора нет аргументов, то это *единица* (одноэлементный тип).

# Алгебраические типы: сумма

- Возможность использовать несколько конструкторов соответствует *сумме* (дизъюнктному объединению).

# Алгебраические типы: сумма

- Возможность использовать несколько конструкторов соответствует *сумме* (дизъюнктному объединению).
- В случае двух типов получаем правила, двойственные произведению:

$$\beta_1 : \forall rs.(r \to r + s) \qquad \beta_2 : \forall rs.(s \to r + s)$$

$$\frac{\Gamma \vdash w : A + B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma \vdash \textbf{match}\ w : (\beta_1 w_1 \Rightarrow u[x := w_1] \mid \beta_2 w_2 \Rightarrow v[y := w_2]) : C}\ \text{Sum}$$

## Алгебраические типы: сумма

- Возможность использовать несколько конструкторов соответствует *сумме* (дизъюнктному объединению).
- В случае двух типов получаем правила, двойственные произведению:

$$\beta_1 : \forall rs.(r \to r + s) \qquad \beta_2 : \forall rs.(s \to r + s)$$

$$\frac{\Gamma \vdash w : A + B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma \vdash \mathbf{match}\ w : (\beta_1 w_1 \Rightarrow u[x := w_1] \mid \beta_2 w_2 \Rightarrow v[y := w_2]) : C}\ \text{Sum}$$

- Как уже говорилось, для расширенной системы типов также работает алгоритм выведения типов.

# Индуктивные типы

- При определении алгебраического типа разрешается также использовать *его самого* в правой части.

# Индуктивные типы

- При определении алгебраического типа разрешается также использовать *его самого* в правой части.
- Пример: тип списка `[a] = [] | a : [a]`.

# Индуктивные типы

- При определении алгебраического типа разрешается также использовать *его самого* в правой части.
- Пример: тип списка `[a] = [] | a : [a]`.
- Алгебраически на это нужно смотреть как на *уравнение*: $X = 1 + A \times X$.

# Индуктивные типы

- При определении алгебраического типа разрешается также использовать *его самого* в правой части.
- Пример: тип списка `[a] = [] | a : [a]`.
- Алгебраически на это нужно смотреть как на *уравнение*: $X = 1 + A \times X$.
- При этом берётся его *наибольшее* (в смысле включения) решение.

# Индуктивные типы

- При определении алгебраического типа разрешается также использовать *его самого* в правой части.
- Пример: тип списка `[a] = [] | a : [a]`.
- Алгебраически на это нужно смотреть как на *уравнение*: $X = 1 + A \times X$.
- При этом берётся его *наибольшее* (в смысле включения) решение.
    - Так, тип `[a]` содержит объект `[]` (пустой список), а также *любой* объект вида `x : xs`, где `x :: a` и `xs :: [a]`.

# Индуктивные типы

- При определении алгебраического типа разрешается также использовать *его самого* в правой части.
- Пример: тип списка `[a] = [] | a : [a]`.
- Алгебраически на это нужно смотреть как на *уравнение*: $X = 1 + A \times X$.
- При этом берётся его *наибольшее* (в смысле включения) решение.
    - Так, тип `[a]` содержит объект `[]` (пустой список), а также *любой* объект вида `x : xs`, где `x :: a` и `xs :: [a]`.
    - Таким образом, тип оказывается не индуктивным, а *коиндуктивным*.

## Индуктивные типы

- При определении алгебраического типа разрешается также использовать *его самого* в правой части.
- Пример: тип списка `[a] = [] | a : [a]`.
- Алгебраически на это нужно смотреть как на *уравнение*: $X = 1 + A \times X$.
- При этом берётся его *наибольшее* (в смысле включения) решение.
  - Так, тип `[a]` содержит объект `[]` (пустой список), а также *любой* объект вида `x : xs`, где `x :: a` и `xs :: [a]`.
  - Таким образом, тип оказывается не индуктивным, а *коиндуктивным.*
  - На практике это означает, что можно построить *бесконечный* объект, однако за счёт ленивости это не страшно.

# Классы типов

- Ещё есть *классы типов:* можно ограничить область значений переменной по типу некоторым классом (например, `Num` — числовые типы).

# Классы типов

- Ещё есть *классы типов:* можно ограничить область значений переменной по типу некоторым классом (например, `Num` — числовые типы).
- Для типов данного класса определены некоторые функции («методы класса»):

```
(+) :: Num a => a -> a -> a
```

# Классы типов

- Ещё есть *классы типов:* можно ограничить область значений переменной по типу некоторым классом (например, `Num` — числовые типы).
- Для типов данного класса определены некоторые функции («методы класса»):

  ```
  (+) :: Num a => a -> a -> a
  ```
- Таким образом реализуется ad hoc полиморфизм: для каждого типа в данном классе своя реализация функции.

- Ещё есть *классы типов:* можно ограничить область значений переменной по типу некоторым классом (например, `Num` — числовые типы).
- Для типов данного класса определены некоторые функции («методы класса»):

  ```
  (+) :: Num a => a -> a -> a
  ```
- Таким образом реализуется ad hoc полиморфизм: для каждого типа в данном классе своя реализация функции.
- Это всё тоже совместимо с выведением типов.

# Параллельные вычисления

- Возвратим небольшой долг — поговорим о *параллельных вычислениях.*

## Параллельные вычисления

- Возвратим небольшой долг — поговорим о *параллельных вычислениях.*
- Речь пойдёт о ситуации, когда мы хотим вычислить «чистую» функцию и ускориться за счёт параллелизма.

## Параллельные вычисления

- Возвратим небольшой долг — поговорим о *параллельных вычислениях.*
- Речь пойдёт о ситуации, когда мы хотим вычислить «чистую» функцию и ускориться за счёт параллелизма.
- Как мы знаем, вычисление в функциональных языках — это последовательность преобразований (редукций) терма.

## Параллельные вычисления

- Возвратим небольшой долг — поговорим о *параллельных вычислениях.*
- Речь пойдёт о ситуации, когда мы хотим вычислить «чистую» функцию и ускориться за счёт параллелизма.
- Как мы знаем, вычисление в функциональных языках — это последовательность преобразований (редукций) терма.
- Теоретически, если в нашем терме есть два независимых подтерма, то их можно редуцировать параллельно (одновременно) на разных вычислительных ядрах.

# Параллельные вычисления

- Возвратим небольшой долг — поговорим о *параллельных вычислениях.*
- Речь пойдёт о ситуации, когда мы хотим вычислить «чистую» функцию и ускориться за счёт параллелизма.
- Как мы знаем, вычисление в функциональных языках — это последовательность преобразований (редукций) терма.
- Теоретически, если в нашем терме есть два независимых подтерма, то их можно редуцировать параллельно (одновременно) на разных вычислительных ядрах.
- С другой стороны, по умолчанию предполагается конкретная *стратегия редукций* (нормальная: редуцируй самый левый редекс).

# Параллельные вычисления

- Таким образом, чтобы активизировать параллельное вычисление, нужно *модифицировать стратегию редукций.*

## Параллельные вычисления

- Таким образом, чтобы активизировать параллельное вычисление, нужно *модифицировать стратегию редукций.*
- Это делается вручную: если автоматически распараллеливать преобразования всех независимых редексов, то расходы на организацию параллельного вычисления превзойдут выгоду от распараллеливания.

## Параллельные вычисления

- Таким образом, чтобы активизировать параллельное вычисление, нужно *модифицировать стратегию редукций.*
- Это делается вручную: если автоматически распараллеливать преобразования всех независимых редексов, то расходы на организацию параллельного вычисления превзойдут выгоду от распараллеливания.
- У нас уже было средство для изменения порядка редукций:

```
seq :: a -> b -> b
```

# Параллельные вычисления

- Таким образом, чтобы активизировать параллельное вычисление, нужно *модифицировать стратегию редукций.*
- Это делается вручную: если автоматически распараллеливать преобразования всех независимых редексов, то расходы на организацию параллельного вычисления превзойдут выгоду от распараллеливания.
- У нас уже было средство для изменения порядка редукций:

```
seq :: a -> b -> b
```

  - Здесь x `` `seq` `` y сначала предвычисляет x, который *может пригодиться* при дальнейшем вычислении y.

# Параллельные вычисления

- Таким образом, чтобы активизировать параллельное вычисление, нужно *модифицировать стратегию редукций.*
- Это делается вручную: если автоматически распараллеливать преобразования всех независимых редексов, то расходы на организацию параллельного вычисления превзойдут выгоду от распараллеливания.
- У нас уже было средство для изменения порядка редукций:
  `seq :: a -> b -> b`
  - Здесь x `` `seq` `` y сначала предвычисляет x, который *может пригодиться* при дальнейшем вычислении y.
- Для параллельного вычисления есть аналогичный
  `par :: a -> b -> b`
  x `` `par` `` y запускает вычисление x параллельно с вычислением y, в надежде, что x пригодится.

# Пример: SAT

- В качестве примера возьмём классическую переборную задачу: поиск выполняющего набора значений переменной для булевой формулы.

# Пример: SAT

- В качестве примера возьмём классическую переборную задачу: поиск выполняющего набора значений переменной для булевой формулы.
- Нужно перебрать $2^n$ наборов, и перебор идеально распараллеливается.

# Пример: SAT

- В качестве примера возьмём классическую переборную задачу: поиск выполняющего набора значений переменной для булевой формулы.
- Нужно перебрать $2^n$ наборов, и перебор идеально распараллеливается.
- Разумеется, неразумно создавать отдельный поток вычисления для каждого набора.

# Пример: SAT

- В качестве примера возьмём классическую переборную задачу: поиск выполняющего набора значений переменной для булевой формулы.
- Нужно перебрать $2^n$ наборов, и перебор идеально распараллеливается.
- Разумеется, неразумно создавать отдельный поток вычисления для каждого набора.
- Мы начнём с того, что распараллелим вычисление на 2 потока: наборы с $p_0 = 0$ и с $p_0 = 1$.

# Пример: SAT

```haskell
data Fm = Var Int | And Fm Fm | Or Fm Fm | Not Fm | Imp Fm Fm

fmEval :: Fm -> [Bool] -> Bool

fmEval (Var n) v = (v !! n)
fmEval (And f1 f2) v = (fmEval f1 v) && (fmEval f2 v)
fmEval (Or f1 f2) v = (fmEval f1 v) || (fmEval f2 v)
fmEval (Not f) v = not (fmEval f v)
fmEval (Imp f1 f2) v = (not (fmEval f1 vals)) || (fmEval f2 v)


myFm n = {- my formula with n+1 variables -}
```

## Пример: SAT

- Список всевозможных наборов из n значений:

```
addtruefalse [] = []
addtruefalse (v : vs) = (True:v) : ((False:v) : addtruefalse vs)
allVals 0 = [[]]
allVals n = addtruefalse (allVals (n-1))
```

# Пример: SAT

- Список всевозможных наборов из n значений:

```
addtruefalse [] = []
addtruefalse (v : vs) = (True:v) : ((False:v) : addtruefalse vs)
allVals 0 = [[]]
allVals n = addtruefalse (allVals (n-1))
```

- Выполняется ли формула на одном из данных наборов?

```
satPartial fm valset = foldr (||) False $ map (fmEval fm) valset
```

## Пример: SAT

- Список всевозможных наборов из n значений:

```
addtruefalse [] = []
addtruefalse (v : vs) = (True:v) : ((False:v) : addtruefalse vs)
allVals 0 = [[]]
allVals n = addtruefalse (allVals (n-1))
```

- Выполняется ли формула на одном из данных наборов?

```
satPartial fm valset = foldr (||) False $ map (fmEval fm) valset
```

- Напомним,

```
map :: (a -> b) -> [a] -> [b]
foldl :: (b -> a -> b) -> b -> [a] -> b
```

# Пример: SAT

- Список всевозможных наборов из n значений:

```
addtruefalse [] = []
addtruefalse (v : vs) = (True:v) : ((False:v) : addtruefalse vs)
allVals 0 = [[]]
allVals n = addtruefalse (allVals (n-1))
```

- Выполняется ли формула на одном из данных наборов?

```
satPartial fm valset = foldr (||) False $ map (fmEval fm) valset
```

- Напомним,

```
map :: (a -> b) -> [a] -> [b]
foldl :: (b -> a -> b) -> b -> [a] -> b
```

- Реализация без распараллеливания (baseline):

```
n = 20
sat = satPartial (myFm n) (allVals (n+1))
main = putStrLn $ show sat
```

# Пример: SAT

- Список всевозможных наборов из n значений:

```
addtruefalse [] = []
addtruefalse (v : vs) = (True:v) : ((False:v) : addtruefalse vs)
allVals 0 = [[]]
allVals n = addtruefalse (allVals (n-1))
```

- Выполняется ли формула на одном из данных наборов?

```
satPartial fm valset = foldr (||) False $ map (fmEval fm) valset
```

- Напомним,

```
map :: (a -> b) -> [a] -> [b]
foldl :: (b -> a -> b) -> b -> [a] -> b
```

- Реализация без распараллеливания (baseline):

```
n = 20
sat = satPartial (myFm n) (allVals (n+1))
main = putStrLn $ show sat
```

- Собираем и запускаем:

```
ghc -threaded -O2 -rtsopts boolean_baseline.hs
./boolean_baseline +RTS -s
```

# SAT без распараллеливания

```
True
     125,939,488 bytes allocated in the heap
         238,120 bytes copied during GC
          77,032 bytes maximum residency (2 sample(s))
          29,120 bytes maximum slop
               2 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       119 colls,     0 par    0.001s   0.001s     0.0000s    0.0000s
  Gen  1         2 colls,     0 par    0.000s   0.000s     0.0001s    0.0002s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.000s elapsed)
  MUT     time    0.636s  (  0.636s elapsed)
  GC      time    0.001s  (  0.001s elapsed)
  EXIT    time    0.000s  (  0.003s elapsed)
  Total   time    0.638s  (  0.640s elapsed)

  Alloc rate    197,932,318 bytes per MUT second

  Productivity  99.7% of total user, 99.3% of total elapsed
```

## SAT в два потока

- Добавим распараллеливание:

```haskell
import Control.Parallel
sat = b `par` (a || b) where
    a = satPartial (myFm n) (map (False:) $ allVals n)
    b = satPartial (myFm n) (map (True:) $ allVals n)
main = putStrLn $ show sat
```

# SAT в два потока

- Добавим распараллеливание:

```haskell
import Control.Parallel
sat = b `par` (a || b) where
    a = satPartial (myFm n) (map (False:) $ allVals n)
    b = satPartial (myFm n) (map (True:) $ allVals n)
main = putStrLn $ show sat
```

- Запускаем:

```
./boolean_par +RTS -s -N2
```

Получаем:

```
 SPARKS: 1 (0 converted, 0 overflowed, 0 dud, 1 GC'd, 0 fizzled)
 Total   time    0.625s  (  0.340s elapsed)
```

# SAT в два потока

- Добавим распараллеливание:

```
import Control.Parallel
sat = b `par` (a || b) where
    a = satPartial (myFm n) (map (False:) $ allVals n)
    b = satPartial (myFm n) (map (True:) $ allVals n)
main = putStrLn $ show sat
```

- Запускаем:

```
./boolean_par +RTS -s -N2
```

Получаем:

```
SPARKS: 1 (0 converted, 0 overflowed, 0 dud, 1 GC'd, 0 fizzled)
Total   time    0.625s  (  0.340s elapsed)
```

- Однако если запустить в один поток (-N1), получается ещё лучше:

```
Total   time    0.309s  (  0.310s elapsed)
```

## SAT в два потока

- Добавим распараллеливание:

```haskell
import Control.Parallel
sat = b `par` (a || b) where
    a = satPartial (myFm n) (map (False:) $ allVals n)
    b = satPartial (myFm n) (map (True:) $ allVals n)
main = putStrLn $ show sat
```

- Запускаем:

```
./boolean_par +RTS -s -N2
```

  Получаем:

```
SPARKS: 1 (0 converted, 0 overflowed, 0 dud, 1 GC'd, 0 fizzled)
Total   time    0.625s  (  0.340s elapsed)
```

- Однако если запустить в один поток (-N1), получается ещё лучше:

```
Total   time    0.309s  (  0.310s elapsed)
```

- Что происходит?

# SAT в два потока

- Дело в ленивости: если мы сумели найти выполняющий набор в первой половине таблицы (`a = True`), то вторую половину (`b`) можно не вычислять.

# SAT в два потока

- Дело в ленивости: если мы сумели найти выполняющий набор в первой половине таблицы (`a = True`), то вторую половину (`b`) можно не вычислять.
- Это и происходит при последовательном вычислении, а при параллельном `b`-поток делает ненужную работу.

# SAT в два потока

- Дело в ленивости: если мы сумели найти выполняющий набор в первой половине таблицы (`a = True`), то вторую половину (`b`) можно не вычислять.
- Это и происходит при последовательном вычислении, а при параллельном `b`-поток делает ненужную работу.
- Это мы и видим в отчёте:

```
SPARKS: 1 (0 converted, 0 overflowed, 0 dud, 1 GC'd, 0 fizzled)
```

# SAT в два потока

- Дело в ленивости: если мы сумели найти выполняющий набор в первой половине таблицы (`a = True`), то вторую половину (b) можно не вычислять.
- Это и происходит при последовательном вычислении, а при параллельном b-поток делает ненужную работу.
- Это мы и видим в отчёте:

```
SPARKS: 1 (0 converted, 0 overflowed, 0 dud, 1 GC'd, 0 fizzled)
```

- Если заменить `||` на `xor` (вычисляем чётность числа выполняющих наборов), то этот эффект исчезает: 0.670s elapsed vs 1.200s elapsed.

# Пример: ParitySAT

ParitySAT с более глубоким распараллеливанием:

```
xsatPartial fm valset = foldr (xor) False $ map (fmEval fm) valset
partValSet m partVals = map (partVals ++) (allVals m)

n = 22
k = 4
m = n + 1 - k

parMapFold f g d [] = d
parMapFold f g d (a : as) = b `par` (bs `g` b) where
    b = f a
    bs = parMapFold f g d as

xsat = parMapFold (xsatPartial (myFm n)) xor False
    (map (partValSet m) (allVals k))

main = putStrLn $ show xsat
```

# Пример: ParitySAT, результаты

| | CPU time | elapsed time | sparks |
|---|---:|---:|---|
| `-N1` | 12.035s | 12.030s | 16 fizzled |
| `-N2` | 11.987s | 6.010s | 8 converted, 8 fizzled |
| `-N4` | 12.753s | 3.250s | 12 converted, 4 fizzled |
| `-N8` | 24.785s | 3.400s | 15 converted, 1 fizzled |