

Лекция 7: вывод типов

Функциональное программирование на Haskell

Алексей Романов

4 апреля 2018

МИЭТ

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):
 - `auto` в C++11
 - `var` в C# 3.0 и Java 10
 - Отсутствие явного типа в Kotlin, Scala

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):
 - `auto` в C++11
 - `var` в C# 3.0 и Java 10
 - Отсутствие явного типа в Kotlin, Scala
- Устроен в них всех похоже:

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):
 - `auto` в C++11
 - `var` в C# 3.0 и Java 10
 - Отсутствие явного типа в Kotlin, Scala
- Устроен в них всех похоже:
 - Тип аргументов функций задаётся явно.

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):
 - `auto` в C++11
 - `var` в C# 3.0 и Java 10
 - Отсутствие явного типа в Kotlin, Scala
- Устроен в них всех похоже:
 - Тип аргументов функций задаётся явно.
 - Дальше типы протягиваются сверху вниз.

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):
 - `auto` в C++11
 - `var` в C# 3.0 и Java 10
 - Отсутствие явного типа в Kotlin, Scala
- Устроен в них всех похоже:
 - Тип аргументов функций задаётся явно.
 - Дальше типы протягиваются сверху вниз.
 - Тип локальных переменных/полей = тип инициализатора (если нет явного).

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):
 - `auto` в C++11
 - `var` в C# 3.0 и Java 10
 - Отсутствие явного типа в Kotlin, Scala
- Устроен в них всех похоже:
 - Тип аргументов функций задаётся явно.
 - Дальше типы протягиваются сверху вниз.
 - Тип локальных переменных/полей = тип инициализатора (если нет явного).
 - Возвращаемый тип = общий тип для всех `return`.

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):
 - `auto` в C++11
 - `var` в C# 3.0 и Java 10
 - Отсутствие явного типа в Kotlin, Scala
- Устроен в них всех похоже:
 - Тип аргументов функций задаётся явно.
 - Дальше типы протягиваются сверху вниз.
 - Тип локальных переменных/полей = тип инициализатора (если нет явного).
 - Возвращаемый тип = общий тип для всех `return`.
- Отдельный вопрос: вывод параметров шаблонов (в C++)/генериков (в других языках).

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов локальных переменных (иногда и возвращаемого типа):
 - `auto` в C++11
 - `var` в C# 3.0 и Java 10
 - Отсутствие явного типа в Kotlin, Scala
- Устроен в них всех похоже:
 - Тип аргументов функций задаётся явно.
 - Дальше типы протягиваются сверху вниз.
 - Тип локальных переменных/полей = тип инициализатора (если нет явного).
 - Возвращаемый тип = общий тип для всех `return`.
- Отдельный вопрос: вывод параметров шаблонов (в C++)/генериков (в других языках).
- Он сложнее, но появился раньше.

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.
- Плюсы:

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.
- Плюсы:
 - Для любого данного выражения есть наиболее общий тип (если есть хоть какой-то).

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.
- Плюсы:
 - Для любого данного выражения есть наиболее общий тип (если есть хоть какой-то).
 - Есть эффективные (для реальных программ) алгоритмы их нахождения.

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.
- Плюсы:
 - Для любого данного выражения есть наиболее общий тип (если есть хоть какой-то).
 - Есть эффективные (для реальных программ) алгоритмы их нахождения.
- Не все расширения Haskell их сохраняют (но это один из критериев оценки расширений).

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.
- Плюсы:
 - Для любого данного выражения есть наиболее общий тип (если есть хоть какой-то).
 - Есть эффективные (для реальных программ) алгоритмы их нахождения.
- Не все расширения Haskell их сохраняют (но это один из критериев оценки расширений).
- Минусы:

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.
- Плюсы:
 - Для любого данного выражения есть наиболее общий тип (если есть хоть какой-то).
 - Есть эффективные (для реальных программ) алгоритмы их нахождения.
- Не все расширения Haskell их сохраняют (но это один из критериев оценки расширений).
- Минусы:
 - Сложнее понять, как работает.

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.
- Плюсы:
 - Для любого данного выражения есть наиболее общий тип (если есть хоть какой-то).
 - Есть эффективные (для реальных программ) алгоритмы их нахождения.
- Не все расширения Haskell их сохраняют (но это один из критериев оценки расширений).
- Минусы:
 - Сложнее понять, как работает.
 - Больше «дальнодействия».

Вывод типов в системе Хиндли-Милнера

- В основе системы типов языка Haskell лежит система Хиндли-Милнера.
- Она изначально создана для вывода типов.
- Даже без указания типов параметров.
- Плюсы:
 - Для любого данного выражения есть наиболее общий тип (если есть хоть какой-то).
 - Есть эффективные (для реальных программ) алгоритмы их нахождения.
- Не все расширения Haskell их сохраняют (но это один из критериев оценки расширений).
- Минусы:
 - Сложнее понять, как работает.
 - Больше «дальнодействия».
- Поэтому понимание работы иногда необходимо.

Постановка задачи

- Даны:
 - Набор типов и конструкторов типов: `Int`, `Bool`, `[]` и т.д.
 - Среди них особую роль играет конструктор функций `->`.
 - Набор известных именованных функций (или констант) с их типами.
 - Выражение, тип которого мы хотим найти (обычно определение функции).

Постановка задачи

- Даны:
 - Набор типов и конструкторов типов: `Int`, `Bool`, `[]` и т.д.
 - Среди них особую роль играет конструктор функций `->`.
 - Набор известных именованных функций (или констант) с их типами.
 - Выражение, тип которого мы хотим найти (обычно определение функции).
- Нужно:
 - Определить наиболее общий тип для этого выражения.

Постановка задачи

- Даны:
 - Набор типов и конструкторов типов: `Int`, `Bool`, `[]` и т.д.
 - Среди них особую роль играет конструктор функций `->`.
 - Набор известных именованных функций (или констант) с их типами.
 - Выражение, тип которого мы хотим найти (обычно определение функции).
- Нужно:
 - Определить наиболее общий тип для этого выражения.
 - Или найти объяснение, почему ему нельзя дать никакой тип.

- Сравните `let x = e1 in e2` и `(\x -> e2) e1`.

- Сравните `let x = e1 in e2` и `(\x -> e2) e1`.
- Значения всегда одинаковы (для любых e_1 и e_2).

- Сравните `let x = e1 in e2` и `(\x -> e2) e1`.
- Значения всегда одинаковы (для любых e_1 и e_2).
- Но есть разница при типизации:

- Сравните `let x = e1 in e2` и `(\x -> e2) e1`.
- Значения всегда одинаковы (для любых e_1 и e_2).
- Но есть разница при типизации: x в лямбде всегда мономорфна, а в `let` может быть полиморфной.

- Сравните `let x = e1 in e2` и `(\x -> e2) e1`.
- Значения всегда одинаковы (для любых e_1 и e_2).
- Но есть разница при типизации: x в лямбде всегда мономорфна, а в `let` может быть полиморфной.
- Поэтому в нетипизированном и просто типизированном λ -исчислении `let` обычно не вводится.

Виды выражений и типов

- В нашей системе 5 видов выражений:

Виды выражений и типов

- В нашей системе 5 видов выражений:
- Названия переменных (в том числе функций) x
- Применение функции к аргументу
- Лямбда-выражения
- `let`-выражения
- `case`-выражения

Виды выражений и типов

- В нашей системе 5 видов выражений:
 - Названия переменных (в том числе функций) x
 - Применение функции к аргументу
 - Лямбда-выражения
 - `let`-выражения
 - `case`-выражения
-
- Типы t :

Виды выражений и типов

- В нашей системе 5 видов выражений:
 - Названия переменных (в том числе функций) x
 - Применение функции к аргументу
 - Лямбда-выражения
 - `let`-выражения
 - `case`-выражения
-
- Типы t :
 - Базовые типы `Int`, ...
 - Переменные типов (греческие буквы α, β, \dots)
 - Тип функций $t_1 \rightarrow t_2$

Виды выражений и типов

- В нашей системе 5 видов выражений:
 - Названия переменных (в том числе функций) x
 - Применение функции к аргументу
 - Лямбда-выражения
 - `let`-выражения
 - `case`-выражения
-
- Типы t :
 - Базовые типы `Int`, ...
 - Переменные типов (греческие буквы α, β, \dots)
 - Тип функций $t_1 \rightarrow t_2$
-
- Полиморфные типы: $\forall \alpha_1 \dots \alpha_k t$

Виды выражений и типов

- В нашей системе 5 видов выражений:
 - Названия переменных (в том числе функций) x
 - Применение функции к аргументу
 - Лямбда-выражения
 - `let`-выражения
 - `case`-выражения
-
- Типы t :
 - Базовые типы `Int`, ...
 - Переменные типов (греческие буквы α, β, \dots)
 - Тип функций $t_1 \rightarrow t_2$
-
- Полиморфные типы: $\forall \alpha_1 \dots \alpha_k t$
 - \forall только на внешнем уровне, под \rightarrow его быть не может!

Унификация

- Нам понадобятся понятия подстановки и унификации. В общем случае:

Унификация

- Нам понадобятся понятия подстановки и унификации. В общем случае:
- Пусть даны два выражения (терма) какого-то формального языка (или несколько пар).
- Они могут содержать переменные.

Унификация

- Нам понадобятся понятия подстановки и унификации. В общем случае:
- Пусть даны два выражения (терма) какого-то формального языка (или несколько пар).
- Они могут содержать переменные.
- Подстановка переменных сопоставляет некоторым переменным выражения (которые тоже могут содержать переменные).

Унификация

- Нам понадобятся понятия подстановки и унификации. В общем случае:
- Пусть даны два выражения (терма) какого-то формального языка (или несколько пар).
- Они могут содержать переменные.
- Подстановка переменных сопоставляет некоторым переменным выражения (которые тоже могут содержать переменные).
- Мы хотим знать, можно ли сделать такую подстановку, чтобы термы стали одинаковыми.

Унификация

- Нам понадобятся понятия подстановки и унификации. В общем случае:
- Пусть даны два выражения (терма) какого-то формального языка (или несколько пар).
- Они могут содержать переменные.
- Подстановка переменных сопоставляет некоторым переменным выражения (которые тоже могут содержать переменные).
- Мы хотим знать, можно ли сделать такую подстановку, чтобы термы стали одинаковыми.
- Это достаточно широко применимое понятие.

Унификация

- Нам понадобятся понятия подстановки и унификации. В общем случае:
- Пусть даны два выражения (терма) какого-то формального языка (или несколько пар).
- Они могут содержать переменные.
- Подстановка переменных сопоставляет некоторым переменным выражения (которые тоже могут содержать переменные).
- Мы хотим знать, можно ли сделать такую подстановку, чтобы термы стали одинаковыми.
- Это достаточно широко применимое понятие.
- Разрешимость задачи унификации и свойства решений зависят от структуры языка.

Унификация типов

- Для вывода типов в системе Хиндли-Милнера случай один из самых простых: унификация первого порядка.
- Подстановки σ сопоставляют переменным типов α, β, \dots *мономорфные* типы (без \forall). Например,

$$\sigma = \{\alpha \mapsto \text{Int}, \beta \mapsto [\gamma]\}$$

- Подстановку можно применить к типу:

$$\sigma(\text{Maybe } \beta) = \text{Maybe } [\gamma]$$

- Композиция подстановок — подстановка.
- Мономорфные типы равны, если они совпадают синтаксически.

Унификация типов

- Для вывода типов в системе Хиндли-Милнера случай один из самых простых: унификация первого порядка.
- Подстановки σ сопоставляют переменным типов α, β, \dots *мономорфные* типы (без \forall). Например,

$$\sigma = \{\alpha \mapsto \text{Int}, \beta \mapsto [\gamma]\}$$

- Подстановку можно применить к типу:

$$\sigma(\text{Maybe } \beta) = \text{Maybe } [\gamma]$$

- Композиция подстановок — подстановка.
- Мономорфные типы равны, если они совпадают синтаксически.
 - Т.е. синонимы типов должны быть уже раскрыты!

Унификация типов

- Тип (α, α) более общий, чем (Int, Int) , так как $\exists \sigma \ \sigma((\alpha, \alpha)) = (\text{Int}, \text{Int})$.
- Второй — частный случай первого.

Унификация типов

- Тип (α, α) более общий, чем (Int, Int) , так как $\exists \sigma \ \sigma((\alpha, \alpha)) = (\text{Int}, \text{Int})$.
- Второй — частный случай первого.
- (α, β) более общий, чем (α, α) .

Унификация типов

- Тип (α, α) более общий, чем (Int, Int) , так как $\exists \sigma \ \sigma((\alpha, \alpha)) = (\text{Int}, \text{Int})$.
- Второй — частный случай первого.
- (α, β) более общий, чем (α, α) .
- Вопрос: когда два типа более общи друг друга?

Унификация типов

- Тип (α, α) более общий, чем (Int, Int) , так как $\exists \sigma \ \sigma((\alpha, \alpha)) = (\text{Int}, \text{Int})$.
- Второй — частный случай первого.
- (α, β) более общий, чем (α, α) .
- Вопрос: когда два типа более общи друг друга?
- Когда они отличаются только названиями переменных.

Унификация типов

- Тип (α, α) более общий, чем (Int, Int) , так как $\exists \sigma \ \sigma((\alpha, \alpha)) = (\text{Int}, \text{Int})$.
- Второй — частный случай первого.
- (α, β) более общий, чем (α, α) .
- Вопрос: когда два типа более общи друг друга?
- Когда они отличаются только названиями переменных.
- Можем рассмотреть задачу унификации $(\alpha, \beta) = (\beta, \text{Int})$.

Унификация типов

- Тип (α, α) более общий, чем (Int, Int) , так как $\exists \sigma \ \sigma((\alpha, \alpha)) = (\text{Int}, \text{Int})$.
- Второй — частный случай первого.
- (α, β) более общий, чем (α, α) .
- Вопрос: когда два типа более общи друг друга?
- Когда они отличаются только названиями переменных.
- Можем рассмотреть задачу унификации $(\alpha, \beta) = (\beta, \text{Int})$.
- $\{\alpha \mapsto \text{Int}, \beta \mapsto \text{Int}\}$ это одно из её решений.

Унификация типов

- Тип (α, α) более общий, чем (Int, Int) , так как $\exists \sigma \ \sigma((\alpha, \alpha)) = (\text{Int}, \text{Int})$.
- Второй — частный случай первого.
- (α, β) более общий, чем (α, α) .
- Вопрос: когда два типа более общи друг друга?
- Когда они отличаются только названиями переменных.
- Можем рассмотреть задачу унификации $(\alpha, \beta) = (\beta, \text{Int})$.
- $\{\alpha \mapsto \text{Int}, \beta \mapsto \text{Int}\}$ это одно из её решений.
- Наиболее общее (все остальные — его частные случаи).

Алгоритм унификации

- Вообще, у каждой задачи унификации типов есть наиболее общее решение (или нет вообще).
- Оно находится следующим алгоритмом:

Алгоритм унификации

- Вообще, у каждой задачи унификации типов есть наиболее общее решение (или нет вообще).
- Оно находится следующим алгоритмом:
- На каждом шаге есть система уравнений. Выбираем любое из них и упрощаем.
- Закончим, когда не останется упрощаемых уравнений (оставшиеся опишут подстановку).
- Правило упрощения зависит от вида рассмотренного уравнения:

Алгоритм унификации: правила

- $C\ t_1\ t_2\ \dots\ t_k = C\ s_1\ s_2\ \dots\ s_k$ (одинаковые конструкторы):

Алгоритм унификации: правила

- $C\ t_1\ t_2\ \dots\ t_k = C\ s_1\ s_2\ \dots\ s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$

Алгоритм унификации: правила

- $C\ t_1\ t_2\ \dots\ t_k = C\ s_1\ s_2\ \dots\ s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$
- $C\ \dots = D\ \dots$ (разные конструкторы):

Алгоритм унификации: правила

- $C t_1 t_2 \dots t_k = C s_1 s_2 \dots s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$
- $C \dots = D \dots$ (разные конструкторы): унификатора не существует.

Алгоритм унификации: правила

- $C\ t_1\ t_2\ \dots\ t_k = C\ s_1\ s_2\ \dots\ s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$
- $C\ \dots = D\ \dots$ (разные конструкторы): унификатора не существует.
- $x = x$:

Алгоритм унификации: правила

- $C t_1 t_2 \dots t_k = C s_1 s_2 \dots s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$
- $C \dots = D \dots$ (разные конструкторы): унификатора не существует.
- $x = x$: оно удаляется, переходим к следующему (можно обобщить и на $t = t$).

Алгоритм унификации: правила

- $C t_1 t_2 \dots t_k = C s_1 s_2 \dots s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$
- $C \dots = D \dots$ (разные конструкторы): унификатора не существует.
- $x = x$: оно удаляется, переходим к следующему (можно обобщить и на $t = t$).
- $x = t$ или $t = x$: два случая.

Алгоритм унификации: правила

- $C t_1 t_2 \dots t_k = C s_1 s_2 \dots s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$
- $C \dots = D \dots$ (разные конструкторы): унификатора не существует.
- $x = x$: оно удаляется, переходим к следующему (можно обобщить и на $t = t$).
- $x = t$ или $t = x$: два случая.
 - Если в t нет переменной x :

Алгоритм унификации: правила

- $C\ t_1\ t_2\ \dots\ t_k = C\ s_1\ s_2\ \dots\ s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$
- $C\ \dots = D\ \dots$ (разные конструкторы): унификатора не существует.
- $x = x$: оно удаляется, переходим к следующему (можно обобщить и на $t = t$).
- $x = t$ или $t = x$: два случая.
 - Если в t нет переменной x : во всех остальных уравнениях делаем замену $x \mapsto t$, переходим к следующему.
 - Если есть

Алгоритм унификации: правила

- $C\ t_1\ t_2\ \dots\ t_k = C\ s_1\ s_2\ \dots\ s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \dots$
- $C\ \dots = D\ \dots$ (разные конструкторы): унификатора не существует.
- $x = x$: оно удаляется, переходим к следующему (можно обобщить и на $t = t$).
- $x = t$ или $t = x$: два случая.
 - Если в t нет переменной x : во всех остальных уравнениях делаем замену $x \mapsto t$, переходим к следующему.
 - Если есть, то унификатора не существует (иначе результат будет бесконечным)!

Алгоритм \mathcal{J} вывода типов

- Γ — окружение типов (набор переменных с их типами).
- $\mathcal{J}(\Gamma; e)$ возвращает наиболее общий тип для e с окружением Γ или выдаёт ошибку.

Алгоритм \mathcal{J} вывода типов

- Γ — окружение типов (набор переменных с их типами).
- $\mathcal{J}(\Gamma; e)$ возвращает наиболее общий тип для e с окружением Γ или выдаёт ошибку.
- ϕ — глобальная переменная (только для простоты объяснения), содержащая уже сделанные подстановки.
- Функция *fresh()* возвращает свежую переменную типа (т.е. такую, которой ещё нигде не было).
- Функция *ftv* возвращает свободные переменные типов в аргументе.
- Функция *unify* решает задачу унификации.

Алгоритм \mathcal{J} : приведение e к стандартному виду

- Переименуем связанные переменные с одинаковыми именами.
- Заменяем `where` на `let`.
- `if` и все сопоставления с образцом в `case`.

Алгоритм \mathcal{J} : рекурсия по определению e

- Переменная x .

Алгоритм \mathcal{J} : рекурсия по определению e

- Переменная x .
 - Тип берётся из окружения (он всегда там есть).
 - Если он полиморфный, то переменные под кванторами заменяются на свежие (новые).

Алгоритм \mathcal{J} : рекурсия по определению e

- Переменная x .
 - Тип берётся из окружения (он всегда там есть).
 - Если он полиморфный, то переменные под кванторами заменяются на свежие (новые).
- Применение функции $e_1 e_2$.

Алгоритм \mathcal{J} : рекурсия по определению e

- Переменная x .
 - Тип берётся из окружения (он всегда там есть).
 - Если он полиморфный, то переменные под кванторами заменяются на свежие (новые).
- Применение функции $e_1 e_2$.
 - Выводим тип t_1 для e_1 .
 - Выводим тип t_2 для e_2 .
 - t_1 должен иметь вид $t_2 \rightarrow \alpha$ (где α — свежая переменная).
Унифицируем их.
 - Результат (тип $e_1 e_2$):

Алгоритм \mathcal{J} : рекурсия по определению e

- Переменная x .
 - Тип берётся из окружения (он всегда там есть).
 - Если он полиморфный, то переменные под кванторами заменяются на свежие (новые).
- Применение функции $e_1 e_2$.
 - Выводим тип t_1 для e_1 .
 - Выводим тип t_2 для e_2 .
 - t_1 должен иметь вид $t_2 \rightarrow \alpha$ (где α — свежая переменная).
Унифицируем их.
 - Результат (тип $e_1 e_2$): α (после подстановок).

Алгоритм \mathcal{J} : рекурсия по определению e

- Лямбда-выражение $\lambda x.e_1$.

Алгоритм \mathcal{J} : рекурсия по определению e

- Лямбда-выражение $\lambda x.e_1$.
 - Тип x — свежая переменная α .
 $x : \alpha$ добавляется в окружение (на время этого вывода).
 - Выводится тип t_1 для e_1 .
 - Результат:

Алгоритм \mathcal{J} : рекурсия по определению e

- Лямбда-выражение $\lambda x.e_1$.
 - Тип x — свежая переменная α .
 $x : \alpha$ добавляется в окружение (на время этого вывода).
 - Выводится тип t_1 для e_1 .
 - Результат: $\alpha \rightarrow t_1$.

Алгоритм \mathcal{J} : рекурсия по определению e

- Лямбда-выражение $\lambda x.e_1$.
 - Тип x — свежая переменная α .
 $x : \alpha$ добавляется в окружение (на время этого вывода).
 - Выводится тип t_1 для e_1 .
 - Результат: $\alpha \rightarrow t_1$.
- let-выражение $\text{let } x = e_1 \text{ in } e_2$

Алгоритм \mathcal{J} : рекурсия по определению e

- Лямбда-выражение $\lambda x. e_1$.
 - Тип x — свежая переменная α .
 $x : \alpha$ добавляется в окружение (на время этого вывода).
 - Выводится тип t_1 для e_1 .
 - Результат: $\alpha \rightarrow t_1$.
- let-выражение $\text{let } x = e_1 \text{ in } e_2$
 - Выводим тип t_1 для e_1 .
 - Тип x получается навешиванием \forall на свободные переменные t_1 , не входящие в окружение.
Добавляем его в окружение

Алгоритм \mathcal{J} : рекурсия по определению e

- Лямбда-выражение $\lambda x. e_1$.
 - Тип x — свежая переменная α .
 $x : \alpha$ добавляется в окружение (на время этого вывода).
 - Выводится тип t_1 для e_1 .
 - Результат: $\alpha \rightarrow t_1$.
- let-выражение $\text{let } x = e_1 \text{ in } e_2$
 - Выводим тип t_1 для e_1 .
 - Тип x получается навешиванием \forall на свободные переменные t_1 , не входящие в окружение. Добавляем его в окружение (тоже на время этого вывода).
 - Выводим тип t_2 для e_2 .
 - Результат:

Алгоритм \mathcal{J} : рекурсия по определению e

- Лямбда-выражение $\lambda x. e_1$.
 - Тип x — свежая переменная α .
 $x : \alpha$ добавляется в окружение (на время этого вывода).
 - Выводится тип t_1 для e_1 .
 - Результат: $\alpha \rightarrow t_1$.
- let-выражение $\text{let } x = e_1 \text{ in } e_2$
 - Выводим тип t_1 для e_1 .
 - Тип x получается навешиванием \forall на свободные переменные t_1 , не входящие в окружение. Добавляем его в окружение (тоже на время этого вывода).
 - Выводим тип t_2 для e_2 .
 - Результат: t_2 .

Алгоритм \mathcal{J} : рекурсия по определению e

- Особый случай: рекурсивный `let` (x входит в e_1).

Алгоритм \mathcal{J} : рекурсия по определению e

- Особый случай: рекурсивный `let` (x входит в e_1).
 - Используем функцию `fix`.
 - `fix f` возвращает неподвижную точку `f` (т.е. `f (fix f) == fix f`).

Алгоритм \mathcal{J} : рекурсия по определению e

- Особый случай: рекурсивный `let` (x входит в e_1).
 - Используем функцию `fix`.
 - `fix f` возвращает неподвижную точку `f` (т.е. `f (fix f) == fix f`).
 - Соответственно, её тип

Алгоритм \mathcal{J} : рекурсия по определению e

- Особый случай: рекурсивный `let` (x входит в e_1).
 - Используем функцию `fix`.
 - `fix f` возвращает неподвижную точку `f` (т.е. `f (fix f) == fix f`).
 - Соответственно, её тип
`fix :: forall a. (a -> a) -> a`.

Алгоритм \mathcal{J} : рекурсия по определению e

- Особый случай: рекурсивный `let` (x входит в e_1).
 - Используем функцию `fix`.
 - `fix f` возвращает неподвижную точку `f` (т.е. `f (fix f) == fix f`).
 - Соответственно, её тип
`fix :: forall a. (a -> a) -> a.`
 - `let x = e1 in e2` превращается в
`let x = fix (\x -> e1) in e2.`
 - Этот `let` не рекурсивный, потому что

Алгоритм \mathcal{J} : рекурсия по определению e

- Особый случай: рекурсивный `let` (x входит в e_1).
 - Используем функцию `fix`.
 - `fix f` возвращает неподвижную точку `f` (т.е. `f (fix f) == fix f`).
 - Соответственно, её тип `fix :: forall a. (a -> a) -> a`.
 - `let x = e1 in e2` превращается в `let x = fix (\x -> e1) in e2`.
 - Этот `let` не рекурсивный, потому что x в `let x = ...` и в `\x -> e1` — разные!

Алгоритм \mathcal{J} : рекурсия по определению e

- case-выражение

case e of

$pat_1 \rightarrow body_1$

$pat_2 \rightarrow body_2$

...

Алгоритм \mathcal{J} : рекурсия по определению e

- case-выражение

case e of

$pat_1 \rightarrow body_1$

$pat_2 \rightarrow body_2$

...

- Выводим тип t для e .
- Для i -той ветви:

Алгоритм \mathcal{J} : рекурсия по определению e

- case-выражение

case e of

$pat_1 \rightarrow body_1$

$pat_2 \rightarrow body_2$

...

- Выводим тип t для e .
- Для i -той ветви:
 - Вводим переменные типов для каждой переменной в pat_i .
 - Выводим тип для pat_i .

Алгоритм \mathcal{J} : рекурсия по определению e

- case-выражение

case e of

$pat_1 \rightarrow body_1$

$pat_2 \rightarrow body_2$

...

- Выводим тип t для e .
- Для i -той ветви:
 - Вводим переменные типов для каждой переменной в pat_i .
 - Выводим тип для pat_i .
 - Унифицируем его с t .

Алгоритм \mathcal{J} : рекурсия по определению e

- case-выражение

case e of

$pat_1 \rightarrow body_1$

$pat_2 \rightarrow body_2$

...

- Выводим тип t для e .
- Для i -той ветви:
 - Вводим переменные типов для каждой переменной в pat_i .
 - Выводим тип для pat_i .
 - Унифицируем его с t .
 - Выводим тип t_i для $body_i$.

Алгоритм \mathcal{J} : рекурсия по определению e

- case-выражение

case e of

$pat_1 \rightarrow body_1$

$pat_2 \rightarrow body_2$

...

- Выводим тип t для e .
- Для i -той ветви:
 - Вводим переменные типов для каждой переменной в pat_i .
 - Выводим тип для pat_i .
 - Унифицируем его с t .
 - Выводим тип t_i для $body_i$.
- Унифицируем все t_i .
- Результат:

Алгоритм \mathcal{J} : рекурсия по определению e

- case-выражение

case e of

$pat_1 \rightarrow body_1$

$pat_2 \rightarrow body_2$

...

- Выводим тип t для e .
- Для i -той ветви:
 - Вводим переменные типов для каждой переменной в pat_i .
 - Выводим тип для pat_i .
 - Унифицируем его с t .
 - Выводим тип t_i для $body_i$.
- Унифицируем все t_i .
- Результат: t_1 (или любой другой t_i).

Примеры

- Разберём несколько примеров

Примеры

- Разберём несколько примеров
- Без `let`:

```
f1 = (.) . (.)
```

Примеры

- Разберём несколько примеров
- Без `let`:

```
f1 = (.) . (.)
```

- С `let`-полиморфизмом:

```
f2 x = let pair x = (x,x) in pair (pair x)
```

Примеры

- Разберём несколько примеров

- Без `let`:

```
f1 = (.) . (.)
```

- С `let`-полиморфизмом:

```
f2 x = let pair x = (x,x) in pair (pair x)
```

- Пример ошибки типа:

```
f3 x = x x
```

Дополнительное чтение

- Более известна альтернатива \mathcal{J} , избегающая «глобальной» ϕ — алгоритм \mathcal{W} . Вы можете легко найти его описания и реализации в Интернете.
- Глава Hindley-Milner Inference в Write You a Haskell
- Algorithm W Step by Step
- Реализация системы типов всего (!) стандарта Haskell 98 с выводом, основанная на \mathcal{J} , есть в Typing Haskell in Haskell.
- Курс лекций В. Н. Брагилевского "Вывод типов от Хиндли — Милнера до GHC 8.8" (есть видео)
- Реализации на Haskell обычно используют монады, но это не должно сильно осложнить понимание.