Лекция 7: вывод типов и ленивость

Функциональное программирование на Haskell

Алексей Романов

4 апреля 2018

МИЭТ

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов:

Вывод типов в других языках

- Во многих ООП-языках сейчас есть вывод типов:
  - `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:
  - `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:
  - `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:
  - `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:
  - `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:
  - `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:
  - `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:
  - `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 констант с их типами (обычно функциями).
  - Выражение, все свободные переменные которого лежат в этом наборе (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 констант с их типами (обычно функциями).
  - Выражение, все свободные переменные которого лежат в этом наборе (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 констант с их типами (обычно функциями).
  - Выражение, все свободные переменные которого лежат в этом наборе (обычно определение функции).
- Нужно:
  - Определить наиболее общий тип для этого выражения.
  - Или найти объяснение, почему ему нельзя дать никакой тип.

`let` и λ

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

`let` и λ

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

`let` и λ

- 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:

`let` и λ

- 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.

`let` и λ

- 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.

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 $\alpha, \beta, \ldots$ *мономорфные* типы. Например,

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

- Подстановку можно применить к типу:

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

- 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 $\alpha, \beta, \ldots$ *мономорфные* типы. Например,

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

- Подстановку можно применить к типу:

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

- 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 $\exists\sigma\ \sigma(($`(`α`, `α`)`$)) =$`(Int, Int)`.
- Второй — частный случай первого.

Унификация типов

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

Унификация типов

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

Унификация типов

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

Унификация типов

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

Унификация типов

- Тип $(\alpha,\ \alpha)$ более общий, чем $(\texttt{Int},\ \texttt{Int})$, так как $\exists\sigma\ \ \sigma((\alpha,\ \alpha)) = (\texttt{Int},\ \texttt{Int})$.
- Второй — частный случай первого.
- Вопрос: когда два типа более общи друг друга?
- Когда они отличаются только названиями переменных.
- Можем рассмотреть задачу унификации $(\alpha,\ \beta) = (\beta,\ \texttt{Int})$.
- $\{\alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{Int}\}$ 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. Одно из них рассматривается (порядок не важен) и система преобразуется.
- Закончим, когда в каждом уравнении слева переменная, среди них нет одинаковых, и справа нет ни одной переменной, которая есть слева (это подстановка).
- Правило выбирается в зависимости от рассмотренного уравнения:

Алгоритм унификации: правила

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

Алгоритм унификации: правила

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

Алгоритм унификации: правила

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

Алгоритм унификации: правила

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

Алгоритм унификации: правила

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

Алгоритм унификации: правила

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

Алгоритм унификации: правила

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

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

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

- $C\ t_1\ t_2\ \ldots\ t_k = C\ s_1\ s_2\ \ldots\ s_k$ (одинаковые конструкторы): заменяется на $t_1 = s_1, t_2 = s_2, \ldots$
- $C\ \ldots = D\ \ldots$ (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.
- Заменим `where` на `let`.
- Все сопоставления с образцом в `case`.
- Заменим рекурсию на вызовы функции `fix :: (a -> a) -> a` и добавим её как константу в Γ.
- `fix f` возвращает неподвижную точку `f` (т.е. `fix f x == x`). Определение:

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

- Переименуем связанные переменные с одинаковыми именами.
- Заменим `where` на `let`.
- Все сопоставления с образцом в `case`.
- Заменим рекурсию на вызовы функции `fix :: (a -> a) -> a` и добавим её как константу в Γ.
- `fix f` возвращает неподвижную точку `f` (т.е. `fix f x == x`). Определение:

```
fix f = let x = f x in x
```

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

- Переименуем связанные переменные с одинаковыми именами.
- Заменим `where` на `let`.
- Все сопоставления с образцом в `case`.
- Заменим рекурсию на вызовы функции `fix :: (a -> a) -> a` и добавим её как константу в Γ.
- `fix f` возвращает неподвижную точку `f` (т.е. `fix f x == x`). Определение:

  ```
  fix f = let x = f x in x
  ```

- Почему это не всегда бесконечный цикл, расскажу в лекции про ленивость. Для вывода типов это неважно.

Алгоритм $\mathcal{J}$: рекурсия по определению e

- $\mathcal{J}(\Gamma; x)$ = do
  $\forall \alpha_1 \dots \alpha_k t = \phi(\Gamma(x))$
  return $t[\alpha_1 \mapsto \mathit{fresh}(), \dots, \alpha_k \mapsto \mathit{fresh}()]$

Алгоритм $\mathcal{J}$: рекурсия по определению e

- $\mathcal{J}(\Gamma; x)$ = do
  $\forall \alpha_1 \ldots \alpha_k t = \phi(\Gamma(x))$
  return $t[\alpha_1 \mapsto \mathit{fresh}(), \ldots, \alpha_k \mapsto \mathit{fresh}()]$

- $\mathcal{J}(\Gamma; e_1\ e_2)$ = do
  $t_1 = \mathcal{J}(\Gamma; e_1)$
  $t_2 = \mathcal{J}(\Gamma; e_2)$
  $\alpha = \mathit{fresh}()$
  $\phi := \mathit{unify}(t_1 = t_2 \to \alpha) \circ \phi$
  return $\phi(\alpha)$

Алгоритм $\mathcal{J}$: рекурсия по определению e

- $\mathcal{J}(\Gamma; \lambda x.e_1)$ = do
  $\alpha = \mathit{fresh}()$
  $t = \mathcal{J}(\Gamma, x : \alpha; e_1)$
  return $\phi(\alpha \to t)$

Алгоритм $\mathcal{J}$: рекурсия по определению e

- $\mathcal{J}(\Gamma; \lambda x.e_1)$ = do
  $\alpha = \mathit{fresh}()$
  $t = \mathcal{J}(\Gamma, x : \alpha; e_1)$
  return $\phi(\alpha \to t)$

- $\mathcal{J}(\Gamma;$ let $x = e_1$ in $e_2)$ = do
  $t_1 = \mathcal{J}(\Gamma; e_1)$
  $\Gamma_1 = \phi(\Gamma)$
  $\mathit{fv} = \mathit{ftv}(t_1) \setminus \mathit{ftv}(\Gamma_1)$
  $t_2 = \forall \mathit{fv}.t_1$
  return $\mathcal{J}(\Gamma_1, x : t_2; e_2)$

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.
- Реализации на Haskell обычно используют монады, но это не должно сильно осложнить понимание. Если не получится, ищите реализации на других языках.
- 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.