Полиморфные типы

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

12 декабря 2019 г.

В этих слайдах

У нас было STLC

- Простое
- Не умеет в рекурсивные функции, следовательно, ограниченное

Где типами T могут быть:

- базовые (ground) типы: $A, B, C, ..., Int, String$
- стрелка между двумя STLC-типами: $t_1 \to t_2$, где $t_1, t_2 \in T$

Некоторые функции (например, `id x = x`) типизируются не типом, а *схемой типов* (например, $\{t_1 \to t_2\}_{t_1, t_2 \in T}$).

Система полиморфных типов Hindley-Milner'a

Типы T включают:

- Множество базовых (ground) типов: $A, B, C, ..., Int, Bool, ...$
- Множетсво типовых переменных: $a, b, c, ...$ (иногда используют греческие буквы)
- Стрелки между двумя типами: $t_1 \to t_2$, где $t_1, t_2 \in T$
- В начале типа может стоять квантор $\forall$ по типовым переменным

Мы будет рассматривать типы, где квантор $\forall$ может стоять только в начале типа, т.е. не встречаться внутри типа.

Вывод типов

Определение (Вывод типов (type inference, type reconstruction))

Процедура построения по данному выражению его типа.

Определение (Проверка типов (type checking))

Процедура проверки, что данное выражение можно протипизировать данным типом.

Рассматриваем prenex-ную форму типов: кванторы всеобщности находится строго впереди типа.
Для prenex-формы и задача проверки типов, и задача вывода типов разрешима.

Полиморфные типы vs. STLC

Все полиморфные типы разбиваются на классы эквивалентности относительно операции переименования типовых переменных.

Пример: выражению `id x = x` можно присвоить и тип `a->a`, и тип `panda -> panda`, но два типа эквивалентны (α-эквивалентны).

Т.е. в STLC выражение `id` типизируется схемой типов, а в полиморфном исчислении – только одним.

Свободные переменные

Левые три как в STLC

$$\begin{aligned} \mathsf{fv}(x) &= x \\ \mathsf{fv}(\lambda x \to e) &= \mathsf{fv}(e) \setminus \{x\} \\ \mathsf{fv}(e_1 e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \end{aligned}$$

Справа – полиморфные типы

$$\begin{aligned} \mathsf{ftv}(\alpha) &= \{\alpha\} \\ \mathsf{ftv}(\tau_1 \to \tau_2) &= \mathsf{ftv}(\tau_1) \cup \mathsf{ftv}(\tau_2) \\ \mathsf{ftv}(\mathsf{Int}) &= \varnothing \\ \mathsf{ftv}(\mathsf{Bool}) &= \varnothing \\ \mathsf{ftv}(\forall x.t) &= \mathsf{ftv}(t) \setminus \{x\} \end{aligned}$$

Подстановки и унификация

Определение (Подстановка полиморфных типов)

Подстановка – конечное отображение из имен типовых переменных в полиморфные типы

Определение (Унификация полиморфных типов)

Унификация двух типов – это поиск такой подстановки, которая после применения к обоим типам даст одинаковые типы.

Пример 1: унификация типов `a -> b` и `Int -> Bool`

Подстановки и унификация

Определение (Подстановка полиморфных типов)

Подстановка – конечное отображение из имен типовых переменных в полиморфные типы

Определение (Унификация полиморфных типов)

Унификация двух типов – это поиск такой подстановки, которая после применения к обоим типам даст одинаковые типы.

Пример 1: унификация типов `a -> b` и `Int -> Bool` завершается успешно с подстановкой $[\mathtt{a} \mapsto \mathtt{Int}, \mathtt{b} \mapsto \mathtt{Bool}]$.

Подстановки и унификация

Определение (Подстановка полиморфных типов)

Подстановка – конечное отображение из имен типовых переменных в полиморфные типы

Определение (Унификация полиморфных типов)

Унификация двух типов – это поиск такой подстановки, которая после применения к обоим типам даст одинаковые типы.

Пример 1: унификация типов `a -> b` и `Int -> Bool` завершается успешно с подстановкой $[\mathtt{a} \mapsto \mathtt{Int}, \mathtt{b} \mapsto \mathtt{Bool}]$.

Пример 2: унификация типов `a -> a` и `Int -> Bool`

Подстановки и унификация

Определение (Подстановка полиморфных типов)

Подстановка – конечное отображение из имен типовых переменных в полиморфные типы

Определение (Унификация полиморфных типов)

Унификация двух типов – это поиск такой подстановки, которая после применения к обоим типам даст одинаковые типы.

Пример 1: унификация типов `a -> b` и `Int -> Bool` завершается успешно с подстановкой $[\mathtt{a} \mapsto \mathtt{Int}, \mathtt{b} \mapsto \mathtt{Bool}]$.

Пример 2: унификация типов `a -> a` и `Int -> Bool` невозможна, так как не существует подстановки, которая бы их сделала одинаковыми.

Формальные правила унификации $a \sim b : \theta$

$$c \sim c : [] \qquad \text{Uni-Const}$$

$$\alpha \sim \alpha : [] \qquad \text{Uni-Var}$$

$$\frac{\alpha \notin \mathsf{ftv}(\tau)}{\alpha \sim \tau : [\alpha/\tau]} \qquad \text{Uni-VarLeft}$$

$$\frac{\alpha \notin \mathsf{ftv}(\tau)}{\tau \sim \alpha : [\alpha/\tau]} \qquad \text{Uni-VarRight}$$

$$\frac{\tau_1 \sim \tau_1' : \theta_1 \qquad [\theta_1]\tau_2 \sim [\theta_1]\tau_2' : \theta_2}{\tau_1\tau_2 \sim \tau_1'\tau_2' : \theta_2 \circ \theta_1} \qquad \text{Uni-Con}$$

$$\frac{\tau_1 \sim \tau_1' : \theta_1 \qquad [\theta_1]\tau_2 \sim [\theta_1]\tau_2' : \theta_2}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2' : \theta_2 \circ \theta_1} \qquad \text{Uni-Arrow}$$

Вывод типов: высокоуровневый наивный алгоритм

Дано: какое-то λ-выражение.

Алгоритм

- Вспомнить все уже известные имена значений и их типы
- Для каждого объявления значения `let x = ... in ...` или `... where x = ...` сгенерировать *ограничения*.
  - к тому, что мы знаем приписываем известные типы (например, `42 :: Int`); иначе приписываем свежую типовую переменную
  - используя "форму" синтаксических выражений создаем ограничения
- Решить полученную систему уравнений, чтобы получить тип значения, которое нам было дано

Вывод типов. Пример (1/2)

```
Prelude> g = \x -> 5+x
g :: Int -> Int
```

Назначим предварительные типы

Подвыражение	Предварительный тип	Собираем ограничения
`\ x -> ((+) 5) x`	`R`	`R = U -> S`
`x`	`U`	
`((+) 5) x`	`S`	
`((+) 5)`	`T`	
`(+)`	`Int -> Int -> Int`	
`5`	`Int`	
`x`	`V`	

Вывод типов. Пример (1/2)

```
Prelude> g = \x -> 5+x
g :: Int -> Int
```

Назначим предварительные типы

Подвыражение	Предварительный тип	Собираем ограничения
`\ x -> ((+) 5) x`	`R`	`R = U -> S`
`x`	`U`	`U = V`
`((+) 5) x`	`S`	
`((+) 5)`	`T`	
`(+)`	`Int -> Int -> Int`	
`5`	`Int`	
`x`	`V`	

Вывод типов. Пример (1/2)

```
Prelude> g = \x -> 5+x
g :: Int -> Int
```

Назначим предварительные типы

Подвыражение	Предварительный тип	Собираем ограничения
`\ x -> ((+) 5) x`	`R`	`R = U -> S`
`x`	`U`	`U = V`
`((+) 5) x`	`S`	
`((+) 5)`	`T`	`Int -> (Int -> Int) = Int -> T`
`(+)`	`Int -> Int -> Int`	
`5`	`Int`	
`x`	`V`	

Вывод типов. Пример (1/2)

```
Prelude> g = \x -> 5+x
g :: Int -> Int
```

Назначим предварительные типы

Подвыражение	Предварительный тип	Собираем ограничения
`\ x -> ((+) 5) x`	`R`	`R = U -> S`
`x`	`U`	`U = V`
`((+) 5) x`	`S`	
`((+) 5)`	`T`	`Int -> (Int -> Int) = Int -> T`
`(+)`	`Int -> Int -> Int`	`T = V -> S`
`5`	`Int`	
`x`	`V`	

Вывод типов. Пример (2/2)

Решаем 4 ограничения:

```
R = U -> S
U = V
Int -> (Int -> Int) = Int -> T
T = V -> S
```

с помощью унификации

- `T = Int -> Int`, подставляя это в 4е ограничение получим
- `Int -> Int = V -> S`, т.е. `Int = V = S`
- Теперь `Int = U`, т.к. `U = V`
- Уточняем первое ограничение: `R = Int -> Int`

Итого, у выражения `g = \x -> 5+x` типом будет `R = Int -> Int`

Ещё примеры/упражнения

- `apply f x = f x`
- `apply g 3`
- `apply not False`

Principal типы

А что если алгоритм вывода типов вывел не подходящий нам тип?

Определение (Наиболее общий унификатор (most general unifier, mgu))

Это такая подстановка-унификатор, что любой другой унификатор получается путём композиции mgu с некоторой подстановкой.

Следствие (Principal типы)

Алгоритм вывода типов выводит наиболее общий (principal) тип.

N.B. Это свойство легко сломать, например, с помощью GADT

Формальные правила типизации

Левые три как в STLC

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \text{T-Var}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad \text{T-App}$$

$$\frac{\Gamma,\ x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x \to e) : \tau_1 \to \tau_2} \quad \text{T-Lam}$$

Правые три – новые

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \quad \text{T-Let}$$

$$\frac{\Gamma \vdash e : \sigma \quad \overline{\alpha} \notin \texttt{ftv}(\Gamma)}{\Gamma \vdash e : \forall\, \overline{\alpha}\ .\ \sigma} \quad \text{T-Gen}$$

$$\frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \quad \text{T-Inst}$$

Новое правило: T-Inst (инстанциация, т.е. уточнение)

$$\frac{\Gamma \vdash e : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2}\ \textsf{T-Inst}$$

Преобразования типа σ в тип τ путём создания свежих имен для каждой типовой переменной, которая не встречается в текущем контексте.

Оператор $\sqsubseteq$ в правиле (T-Inst) означает, что тип является конкретизацией схемы типов.

$$\forall a.a \to a \sqsubseteq \mathsf{Int} \to \mathsf{Int}$$
$$\forall a.a \to a \sqsubseteq b \to b$$
$$\forall ab.a \to b \to a \sqsubseteq \mathsf{Int} \to \mathsf{Bool} \to \mathsf{Int}$$

Новое правило: T-Gen (generalization, т.е. обобщение)

$$\frac{\Gamma \vdash e : \sigma \quad \overline{\alpha} \notin \texttt{ftv}(\Gamma)}{\Gamma \vdash e : \forall\ \overline{\alpha}\ .\ \sigma}\ \text{T-Gen}$$

Пример: `id = \ x -> x` в контексте $\Gamma = \varnothing$

$$\frac{\vdash id : (a \rightarrow a) \quad a \notin \varnothing}{\vdash id : \forall a\ .\ (a \rightarrow a)}$$

Новое правило: T-Let – let-полиморфизм (1/2)

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}\ \text{T-Let}$$

Пример:

```
let double f z = f (f z) in
(double (\ x -> x+1) 1, double (\ x -> not x) false)
```

Выведенный тип для `f` в функции `double` мог бы быть `x -> x`. В алгоритме выше использование `double` на первом аргумента породит ограничение `x = Int`, а второе использование `double` породит ограничение `x = Bool`. Эти ограничения несовместны, потому могли бы привести к невозможности унификации.

Поэтому при реализации клонирует тип `let`-выражения, чтобы типы не "склеились"

Новое правило: T-Let – `let`-полиморфизм (2/2)

При исполнении кода

```
let double f z = f (f z) in
(double (\ x -> x+1) 1, double (\ x -> not x) false)
```

можно исполнять такой код

```
(\double -> ( double (\ x -> x+1)   1
            , double (\ x -> not x) false) )
   (\f z -> f (f z))
```

Но нижний код не типизируется в системе полиморфных типов Хиндли-Милнера, а тот, что выше – типизируется.

Occurs check

Дополнительная проверка, которая объявляет некоторые подстановки некорректными.

Чтобы было проще применять подстановку σ, мы можем подставить переменные из $dom(\sigma)$ в правые части подстановки. Это удастся, если в подстановке *нет циклов*.

Пример: `let f x = f` отклоняется Haskell

```
<interactive>:1:1: error:
    • Occurs check: cannot construct the infinite type: t ~ p0 -> t
    • Relevant bindings include f :: t (bound at <interactive>:1:1)
```

Потому, что мы не может сунифицировать типы `b` и `a->b` так, чтобы получился конечный тип

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

<interactive>:3:21: error:
    • Occurs check: cannot construct the infinite type: t0 ~ t0 -> t
      Expected type: t0 -> t
        Actual type: (t0 -> t) -> t
    • In the first argument of ‘x’, namely ‘x’
      In the first argument of ‘f’, namely ‘(x x)’
      In the expression: f (x x)
    • Relevant bindings include
        x :: (t0 -> t) -> t (bound at <interactive>:3:11)
        f :: t -> t (bound at <interactive>:3:5)
        fix :: (t -> t) -> t (bound at <interactive>:3:1)
```

В то время как

```
Prelude> fix f = f (fix f)
Prelude> :t fix
fix :: (t -> t) -> t
```

Но если Haskell что-то не умеет, то это не значит, что никто не умеет

```
$ ocaml -rectypes -noinit
        OCaml version 4.07.1

# let fix f = (fun x -> f (x x)) (fun x -> f (x x));;
val fix : ('a -> 'a) -> 'a = <fun>
```

Ссылки I

- Hindley-Milner type inference in Haskell
  Stephen Diehl
  blog post
- Lecture notes on type inference from Cornell University
  URL