Объектно-ориентированное программирование и ТТ

Отношение «быть подтипом»

Определение

Будем говорить, что A — подтип B:

$$A <: B$$

если значения типа A — часть множества значений типа B.

Пример

- *В Java String $<:$ Object.*
- *В Java int $\not<:$ Integer.*

Ко- и контравариантность

Определение взято из теории категорий (и упрощено):

Определение

Пусть заданы два упорядоченных множества, $\langle C, < \rangle$ *и* $\langle D, < \rangle$*. Ковариантное (контравариантное) отображение —* $f : C \to D$*, что*

- *при* $x < y$ *всегда* $f(x) < f(y)$ *(ковариантное отображение);*
- *при* $x < y$ *всегда* $f(y) < f(x)$ *(контравариантное отображение).*

Операции на типах

Пусть $g : \star \to \star$. Что можно сказать о его ко- или контравариантности?

- Функции ковариантны по результату: $g(\sigma) := \mathsf{int} \to \sigma$.
  Если $\alpha <: \beta$, $a : g(\alpha)$, $b : g(\beta)$ то $a(n) : \alpha, b(n) : \beta$, поэтому функцию b можно всегда заменить на a. Отсюда,
  $$\alpha <: \beta \text{ влечёт } g(\alpha) <: g(\beta)$$
- Функции контравариантны по аргументу: $g(\sigma) := \sigma \to \mathsf{int}$.
  Если $\alpha <: \beta$, $a : g(\alpha)$, $b : g(\beta)$, и $x : \alpha$ то $a(x)$ и $b(x)$ законны. Отсюда,
  $$\alpha <: \beta \text{ влечёт } g(\beta) <: g(\alpha)$$
- Массивы инвариантны: $g(\alpha) := \alpha[]$.
  С одной стороны, $g(\alpha).set : \alpha \to ()$, с другой стороны, $g(\alpha).get : \mathsf{int} \to \alpha$.
  В Java массивы ковариантны, что приводит к проверке времени исполнения.

Структурная и именная эквивалентность

```
struct X { a: int, b: char } x;
struct Y { a: int, b: char } y;
```

Структурная эквивалентность: $X \approx Y$
Именная эквивалентность: $X \not\approx Y$

Исчисление $F_{<:}$

Язык:

$$T ::= x | \lambda x^{\tau}.T | T\ T | \lambda\alpha <: \tau.T | t\ \tau$$

Типы:

$$\tau ::= \alpha | \top | \tau \to \tau | \forall\alpha <: \tau.\tau$$

Подтипизация:

$$\frac{}{\Gamma \vdash \sigma <: \sigma} \qquad \frac{\Gamma \vdash \sigma <: \rho \qquad \Gamma \vdash \rho <: \tau}{\Gamma \vdash \sigma <: \tau} \qquad \frac{}{\Gamma \vdash \sigma <: \top}$$

$$\frac{}{\Gamma, \alpha <: \tau \vdash \alpha <: \tau} \qquad \frac{\Gamma \vdash \tau_1 \to \sigma_1 \qquad \Gamma \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}$$

Исчисление $F_{<:}$

Типизация: правила системы F со следующими добавлениями/изменениями

$$\frac{\Gamma, \alpha <: \tau_1 \vdash T_2 : \tau_2}{\Gamma \vdash \lambda\alpha <: \tau_1 . T_2 : \forall\alpha <: \tau_1 . \tau_2} \text{ тип. абстракция}$$

$$\frac{\Gamma, T_1 : \forall\alpha <: \tau_{11} . \tau_{12} \qquad \Gamma \vdash \tau_2 <: \tau_{11}}{\Gamma \vdash T_1\ \tau_2 : \tau_{12}[\alpha := \tau_2]} \text{ тип. применение}$$

$$\frac{\Gamma \vdash T : \sigma \qquad \Gamma \vdash \sigma <: \tau}{\Gamma \vdash T : \tau} \text{ приведение}$$

Полное и ядерное правила

Ядерное правило подтипизации:

$$\frac{\Gamma, \alpha <: \rho_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \forall \alpha <: \rho_1 . \sigma_2 <: \forall \alpha <: \rho_1 . \tau_2}$$

Полное правило подтипизации:

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \qquad \Gamma, \alpha <: \tau_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \forall \alpha <: \sigma_1 . \sigma_2 <: \forall \alpha <: \tau_1 . \tau_2}$$

Типизация для пар

Можно показать:

$$\frac{\Gamma \vdash \sigma_1 <: \tau_1 \qquad \Gamma \vdash \sigma_2 <: \tau_2}{\sigma_1 \& \sigma_2 <: \tau_1 \& \tau_2}$$

Что такое класс/объект

- Определим отношение подтипизации на кортежах — это даст наследование.

$$\text{struct } x_1 : \tau_1, \ldots, x_n : \tau_n \text{ end} ::= \langle x_1 : \tau_1, \langle \ldots \langle x_n : \tau_n, T : \top \rangle \rangle$$

  Упростим, убрав имена полей (не теряя общности):

$$\text{struct } \tau_1, \ldots, \tau_n \text{ end} ::= \tau_1 \& \ldots (\tau_n \& \top)$$

  Тогда

$$\tau_n \& \tau_{n+1} <: \tau_n \& \top$$

  Отсюда,

$$\text{struct } x_1 : \tau_1, \ldots, x_n : \tau_n \text{ end} <: \text{struct } x_1 : \tau_1, \ldots, x_{n-1} : \tau_{n-1} \text{ end}$$

- Приведения типов и т.п. получаются согласно общим правилам $F_{<:}$.
- По необходимости добавим экзистенциальные типы.
- При необходимости сделать именную эквивалентность из структурной можно, включив какие-нибудь токены в тип .

Objective Caml

- ML — Meta Language, Робин Милнер, 1970е.
- Standard ML, 1983.
- Category Abstract Machine Language (Caml), 1985.
- ZINC project (ZINC Is Not CAML): “Toplevels considered harmful”, Ксавье Леруа (Xavier Leroy), 1990.
- Objective Caml, 1996.

Объекты, классы, подтипы

- Объектно-ориентированность — набор различных конструкций, которые можно собирать по-разному.
- Объектно-ориентированность в Окамле собрана иначе, чем в Джаве или в Смолтоке.
- В Окамле есть две различных конструкции: модули (соответствуют абстрактным типам данных), и объекты и классы (предназначены для построения иерархии наследования).
- Отношение подтипирования определено независимо от модулей и классов, в том числе и на обычных типах.

Полиморфный вариантный тип

Традиционный алгебраический тип не допускает пересечение вариантов:

```
type abc = A | B | C;;
type ac = A | C;;                (* ac.C <> abc.C *)
```

Однако, есть полиморфный вариантный тип:

```
type abc = [‘A | ‘B | ‘C];;
type ac = [‘A | ‘C ];;
```

Заметим, что $[> `A | `C] :> [> `A | `B | `C]$.

Автоматический вывод типов не может построить правильное подтипирование, но можно указать его явно:

```
let f a = ‘A;;                   (* f : ’a -> [> ‘A ] *)
let g = (f () : ac);;            (* ошибка: у типа f() нет тэга ‘C *)
let g = (f () : [‘A] :> ac);;    (* g : ac, явное приведение типа *)
```

Модули: ковариантный интерфейс

```
module type Writer = sig
    type +'b t
    val empty : unit -> 'b t
    val push : 'b t -> 'b -> 'b t
    val print_len : 'b t -> unit
end

module Writer : Writer = struct
    type 'b t = 'b list
    let empty () = []
    let push l x = x :: l
    let print_len l = print_int (List.length l)
end

let w = Writer.empty ();;
let w = Writer.push w `A;;
let w = Writer.push w `C;;
let u = (w : [`A|`C] Writer.t :> [`A|`C|`E] Writer.t);;
let u = Writer.push u `E;;
```

Модули: контравариантный интерфейс

```
module type Counter = sig
    type -'b t
    val empty : ('b->int->int) -> 'b t
    val push : 'b t -> 'b -> 'b t
    val print_len : 'b t -> unit
end

module Counter : Counter = struct
    type 'b t = int * ('b->int->int)
    let empty f = (0,f)
    let push (l,f) x = (f x l, f)
    let print_len (l,f) = print_int l
end

let w = Counter.empty (fun x l -> l + match x with `A -> 0 | `B -> 1 | `C -> 2);;
let w = Counter.push w `A;;
let w = Counter.push w `C;;
let u = (w : [`A|`C] Counter.t :> [`A] Counter.t);; (* однако, [`A] :> [`A|`C] *)
let u = Counter.push u `A;;
```

Объекты

Зададим объект:

```
type square = < area : float; width : int >;;

let square w = object
  method area = Float.of_int (w * w)
  method width = w
end;;
```

И нечто общее:

```
let coin = object
  method shape = circle 5
  method color = "silver"
end;;

let map = object
  method shape = square 10
end;;

type item = < shape : shape >;;
let items = [ (coin :> item) ; (map :> item) ];;
```

Формализация: подтипы

- "Быть подтипом" определяется рекурсивно — согласно структуре.
- Приведение типа:

$$\frac{\tau <: \tau' \qquad \Gamma \vdash \alpha : \theta(\tau)}{\Gamma \vdash (\alpha : \tau <: \tau') : \theta(\tau')}$$

Формализация: экзистенциальные типы и модули

- ```
  module type Counter = sig
      type -'b t
      val empty : ('b->int->int) -> 'b t
      ...
  end
  ```
  То есть, $\exists \beta . E : (\beta \to N \to N) \to \tau(\beta)$
- Заменим экзистенциальный тип (интерфейс модуля) на его каноническое представление: $(B \to N \to N) \to \tau(B)$
- Правила подтипирования для типа, на основании его канонического представления:
  $$\frac{\Gamma, \alpha \vdash R <: R'[\alpha' := \tau]}{\Gamma \vdash \exists \alpha . R <: \exists \alpha' . R}$$
  ```