Controllo dei tipi

Linguaggi e tipi

- Il problema che ci poniamo ora è quello di comprendere come la struttura dei tipi del linguaggio possa influenzare il progetto e la struttura di implementazione dei linguaggi di programmazione.
- Gli interpreti che abbiamo considerato effettuano il controllo dei tipi a tempo di esecuzione

Sistema dei tipi

- Un **sistema dei tipi** associa *tipi* a ogni valore calcolato.
- Esaminando il flusso dei valori calcolati, un sistema dei tipi tenta di dimostrare che non avvengano *errori di tipo*.
- Il sistema stesso determina che cosa costituisce un errore di tipo, garantendo che le operazioni che si aspettano un certo tipo di valore non siano utilizzate con valori per i quali quell'operazione non ha senso.

Un esempio

- Consideriamo l'espressione
  let x = e1 in e2
- Quale è il tipo associato a questa espressione?
- Quale è il tipo della variabile dichiarata nel blocco let?
- Occorrenze differenti della variabile 'x' possono essere associate a tipi differente in considerazione delle regole di scoping.
- Problema: tenere traccia dei tipi associati alle variabili

Ambiente dei tipi

- L'ambiente dei tipi è una funzione (di dominio finito) che associa identificatori a tipi.
- Noi scriveremo:

$$\boldsymbol{\Gamma = x_1 : \tau_1, x_2 : \tau_2 \ \ldots x_k : \tau_k}$$

- Per indicare la funzione

$$\boldsymbol{\Gamma(x_i) = \tau_i}$$

- che associa il tipo τ_i al valore x_i
- Useremo

$$\boldsymbol{\Gamma, x : \tau}$$

- per indicare l'estensione della funzione Γ con l'associazione $x : \tau$
  - $(\Gamma, \mathrm{x} : \tau)(\mathrm{x}) = \tau$ e $(\Gamma, \mathrm{x} : \tau)(\mathrm{y}) = \Gamma(y)\ per\ y\ != x$

Esempi

$$\Gamma = x : \tau, y : \tau'$$
$$\Gamma(x) = \tau$$
$$\Gamma(z) = undefined$$
$$\Gamma' = \ \Gamma, \mathrm{z} : \tau_2$$
$$\Gamma'(z) = \tau_2$$

Assegnamento di tipo

Supponiamo che Γ sia un ambiente di tipo.

$$\boxed{\boldsymbol{\Gamma \vdash e : \tau}}$$

Per indicare che l'espressione e ha tipo τ nell'ambiente di tipo Γ.

Sistema di tipo

Definiamo il sistema per il controllo dei tipi (type checker) per il linguaggio funzionale didattico (senza funzioni)

$$\boxed{\Gamma \vdash n : int}$$

$$\boxed{\Gamma \vdash b : bool}$$

$$\boxed{\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}}$$

Blocco let

$$\frac{\Gamma \vdash e_1 : \tau_1, \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau_2}$$

Operatori binari e condizionale

$$\frac{\Gamma \vdash e_1 : \tau_1, \Gamma \vdash e_2 : \tau_2, \quad \oplus : \tau_1 \times \tau_2 \to \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

$$\frac{\Gamma \vdash e : bool, \Gamma \vdash e_1 : \tau, \Gamma \vdash e_2 : \tau}{\Gamma \vdash if\ e\ then\ e_1\ else\ e_2 : \tau}$$

Esempi

$$\dfrac{\dfrac{}{\vdash 1 : \texttt{int}} \quad \dfrac{\dfrac{}{x : \texttt{int} \vdash x : \texttt{int}} \quad \dfrac{}{x : \texttt{int} \vdash 1 : \texttt{int}}}{x : \texttt{int} \vdash x + 1 : \texttt{int}}}{\vdash \texttt{let}\ x = 1\ \texttt{in}\ x + 1 : \texttt{int}}$$

$$\dfrac{\dfrac{}{\vdash 1 : \texttt{int}} \quad \dfrac{x : \texttt{int} \vdash x : \texttt{bool} \quad \cdots}{x : \texttt{int} \vdash \texttt{if}\ x\ \texttt{then}\ 42\ \texttt{else}\ 17 : ??}}{\vdash \texttt{let}\ x = 1\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ 42\ \texttt{else}\ 17 : ??}$$

Tipi per le funzioni

- Per semplicità consideriamo funzioni con un solo argomento non ricorsive (in sintassi concreta fun x -> e).
- Quale è il tipo di una funzione?
- Il costruttore di tipo τ1 → τ2 descrive il tipo della funzioni che prendono in ingress un argomento di tipo τ1 e restituiscono un risultato di tipo τ2”

Funzioni: Controllo di tipo

DEFINIZIONE

$$\frac{\Gamma, x: \tau_1 \vdash e: \tau_2}{\Gamma \vdash fun(x: \tau_1) = e: \tau_1 \to \tau_2}$$

INVOCAZIONE

$$\frac{\Gamma \vdash e_1: \tau_1 \to \tau_2, \Gamma \vdash e_2: \tau_1}{\Gamma \vdash App(e_1, e_2): \tau_2}$$

Controllo dei tipi e Regole di Scope

- Consideriamo il programma:
  e = let x = 1 in
  let f = fun (y : int) = x + y in
  let x = 10 in f (3)
- Applicando le regole di type checking abbiamo che

$$\frac{\frac{\Gamma = x: int, y: int \vdash x + y: int}{x: int \vdash fun(y: int) = x + y: int \to int}}{x: int \vdash let\ f = fun(y: int) = x + y: int \to int}$$

Controllo dei tipi e regole di scoping

```
e = let x = 1 in
  let f = fun (y : int) = x + y in
    let x = 10 in f (3)
```

$\Gamma = x: int, f: int \to int$

$$\dfrac{\dfrac{}{x: \boldsymbol{int} \vdash \mathbf{1}: \boldsymbol{int}} \quad \dfrac{x: \boldsymbol{int} \vdash \boldsymbol{let}\ f = \boldsymbol{fun}(y: \boldsymbol{int}) = x + y: \boldsymbol{int} \to \boldsymbol{int}, \dfrac{\Gamma \vdash \mathbf{10}: \boldsymbol{int}, \Gamma \vdash f(\mathbf{3}): \boldsymbol{int}}{\Gamma \vdash \boldsymbol{let}\ x = \mathbf{10}\ \boldsymbol{in}\ f(\mathbf{3}): \boldsymbol{int}}}{x: \boldsymbol{int} \vdash \text{let f = fun (y : int) = x + y in let x = 10 in f (3)}}}{\emptyset \vdash \boldsymbol{e}: \boldsymbol{int}}$$

Controllo dei tipi e regole di scope

- Quando applichiamo le regole di typechecking, la funzione f è una funzione da interi a interi (ha tipo int -> int), x è un valore intero, quindi il programma supera il controllo dei tipi ma restituice un risultato differente a seconda delle regole di scope:
- Static scope: 1+3 = 4
- Dynamic scope: 10+ 3 = 13
- Ma potrebbe andare anche peggio!!

Controllo tipi e regole di scope

- Consideriamo il programma e assumiamo lo scoping dinamico
  e = let x = 1 in
  let f = fun (y : int) = x + y in
  let x = true in f (3)
- Le regole di tipo associano il tipo int -> int alla funzione f, ma prima dell'invocazione della funzione la variabile x viene legata a un valore di tipo bool (viene pertanto generato un errore a run-time)
- Morale: *lo scoping dinamico non si amalgama bene con le regole di type checking*.

Nota

- Funzioni anonime (come Fun(x: τ)=e del linguaggio didattico) sono presenti in molti linguaggi di programmazione, ad esempio Java a partire dalla versione 8).
- Possiamo introdurre funzioni anonime dette (lambda) con la motazione:
  λx : τ. e
  La notazione λ è stata introdotta dal logico Church (1940): λ-calcolo
- Funzioni anonime in
  - Scala: (x: Type) => e
  - Java 8: x -> e (senza dichiarare il tipo)
  - Haskell: \x -> e oppure \x::Type -> e

Esempio: Python

```
# Python code to illustrate cube of a number
# showing difference between def() and lambda().

def cube(y):
    return y*y*y;

g = lambda x: x*x*x

print(g(7))

print(cube(5))
```

Programmiamo un type checker in OCAML

Il primo passo metodologico: tipi e ambiente

- La struttura dei tipi

```
type tval = TInt | TBool | FunT of tval * tval
```

- La definizione dell'ambiente dei tipi

```
type tenv = ide -> tval;;
let bind (r :  tenv) (i : ide) (v : tval) =
   function x -> if x = i then v else r x;;
let tenv0 = fun (x: ide) -> raise EmptyEnv;;
```

La sintassi delle espressioni con tipi

```
type texp = EInt of int
      | EBool of bool
      | Den of ide
      | Add of texp * texp
      | Sub of texp * texp
      | Not of texp
      | And of texp * texp
      | Ifz of texp * texp * texp
      | Eq of texp * texp
      | Let of ide * texp * texp
      | Fun of ide * tval * texp
      | App of texp * texp
```

Dalle regole al type checker teval

let rec teval e tenv =
match e with

$\boxed{\Gamma \vdash n: int}$ **EInt i -> TInt**

$\boxed{\Gamma \vdash b: bool}$ **| EBool b -> TBool**

$\boxed{\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x: \tau}}$ **| Den s -> tenv s**

Il blocco let

$$\boxed{\frac{\boldsymbol{\Gamma \vdash e_1: \tau_1, \Gamma, x: \tau_1 \vdash e_2: \tau_2}}{\boldsymbol{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau_2}}}$$

Let(i, e1, e2) -> let t = teval e1 tenv in
teval e2 (bind tenv i t)

Operatori binari

$$\frac{\Gamma \vdash e_1 : \tau_1, \Gamma \vdash e_2 : \tau_2, \quad \oplus : \tau_1 \times \tau_2 \to \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

```
Add (e1,e2) -> let t1 = teval e1 tenv in
                 let t2 = teval e2 tenv in
                  ( match (t1, t2) with
                    (TInt, TInt) -> TInt
                   | (_,_)  -> raise WrongType )
```

```
Eq (e1,e2) ->
     if ((teval e1 tenv) = (teval e2 tenv))
       then TBool
         else raise WrongType
```

Condizionale

$$\frac{\Gamma \vdash e : \mathbf{bool}, \Gamma \vdash e_1 : \tau, \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : \tau}$$

```
Ifz (e1,e2,e3) -> let t = teval e1 tenv in
                    ( match t with
                        TBool -> let t1 = teval e1 tenv in
                                   let t2 = teval e2 tenv in
                                    ( match (t1, t2) with
                                       | (TInt, TInt) -> TInt
                                       | (TBool, TBool) -> TBool
                                       | (_,_) -> raise WrongType )
                       | _ -> raise WrongType )
```

Funzioni

$$\frac{\Gamma, x{:}\,\tau_1 \vdash e{:}\,\tau_2}{\Gamma \vdash fun(x{:}\,\tau_1) = e{:}\,\tau_1 \to \tau_2}$$

```
Fun(i, t1, e) -> let tenv1 = bind tenv i t1 in
                 let t2 = teval e tenv1 in FunT(t1,t2)
```

$$\frac{\Gamma \vdash e_1{:}\,\tau_1 \to \tau_2, \Gamma \vdash e_2{:}\,\tau_1}{\Gamma \vdash App(e_1, e_2){:}\,\tau_2}$$

```
App(e1, e2) ->
    let f = teval e1 tenv in
        ( match f with
          FunT(t1,t2) -> if  ((teval e2 tenv) = t1 )
                             then t2 else raise WrongType
        | _ -> raise WrongType )
```

Esempio

```
let ff = Let ("x", EInt 1,
   Let ("f", Fun ("y", TInt, Add (Den "x", Den "y")),
    Let ("x", EBool true, App (Den "f", EInt 3))))
```

```
# teval ff tenv0;;
- : tval = TInt
```

La ricorsione

Funzioni ricorsive

- Estendiamo il linguaggio con un meccanismo per definire funzioni ricorsive

  e::= …. **rec f(x:τ_1):τ_2 = e**

- Possiamo definire let rec come zucchero sintattico:
- **Letrec f(x:τ_1):τ_2 = e_1 in e_2 ⇐⇒ let f = rec f(x:τ_1):τ_2= e_1 in e_2**

Regola di tipo

RiCORSIONE

$$\frac{\Gamma, f: \tau_1 \to \tau_2, x: \tau_1 \vdash e: \tau_2}{\Gamma \vdash rec\ f(x: \tau_1): \tau_2 = e: \tau_1 \to \tau_2}$$

Si estende l'ambiente dei tipo con il tipo della funzione ($\tau_1 \to \tau_2$) e il tipo dell'argomento (τ_1)e poi si applicano le regole di tyoe checking per il corpo della funzione ricorsiva e

Esercizio

- Estendere il codice OCAML del typechecker **teval** per gestire le funzioni ricorsive

Pair

- Coppie di valori (**Pair**): il modo immediato di combinare dati
  - **(1, 2), (true, false), (1, (true, fun(x:int) = x + 2))**
- Estrarre le componenti di una coppia:
  - **fst (1, 2) -> 1 snd (true, false) -> false**
- Pattern matching per estrarre le componenti di una coppia
  - **let pair (x,y) = (1,2) in (y,x) -> (2,1)**

Pair nei linguaggi di programmazione

Haskell	Scala	Java	Python
`(1,2)`	`(1,2)`	`new Pair(1,2)`	`(1,2)`
`fst e`	`e._1`	`e.getFirst()`	`e[0]`
`snd e`	`e._2`	`e.getSecond()`	`e[1]`
`let (x,y) =`	`val (x,y) =`	N/A	N/A

Sintassi

e ::= ⋯|(e1,e2)|fst e|snd e
| let pair (x,y) = e1 in e2

Regole di valutazione

$$\frac{env \rhd e_1 \Rightarrow v_1, env \rhd e_2 \Rightarrow v_2}{env \rhd (e_1, e_2) \Rightarrow (v_1, v_2)}$$

$$\frac{env \rhd e \Rightarrow (v_1, v_2)}{env \rhd fst(e) \Rightarrow v_1} \qquad \frac{env \rhd e \Rightarrow (v_1, v_2)}{env \rhd snd(e) \Rightarrow v_2}$$

$$\frac{env \rhd e \Rightarrow (v_1, v_2), e_2[v_1/x,\ v_2/y] \Rightarrow v}{env \rhd let\ pair\ (x, y) = e_1\ in\ e_2 \Rightarrow v}$$

Regole di tipo

$$\frac{\Gamma \vdash e_1 : \tau_1, \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash fst\ e \vdash \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash snd\ e \vdash \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2, \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash let\ pair(x, y) = e_1\ in\ e_2 : \tau}$$

Esercizio

- Definire interprete e type checker per il linguaggio didattico esteso con il costrutture di tipo ***pair***

Dalle coppie alle tuple e…

- Possiamo aggiungere costruttori per triple, quadrule, ennuple di valori: le tuple.
- Esempio (Python)
  - thistuple = ("apple", "banana", "cherry")
  - thistuple = ("apple", "banana", "cherry ")
    print(thistuple[1])
    >> banana

Valori null

- Vogliamo avere un valore speciale "**null**" per indicare l'assenza di un valore
- In Java il valore **null** è il valore di default degli oggetti
- Come abbiamo visto nella prima parte del corso questo ha portato alla definizione della strategoa di propgrammazione difensiva per evitare NullPointerException in Java (meccanismo delle exception)

Possiamo fare di meglio?

- Option type
  - **e ::= ….. | none | some (e)**
  - **τ ::= …. | option[τ]**
- Il valore none viene utilizzato per esprimere l'assenza di un valore mentre some(e) esprime la presenza del valore (risultato della valutazione di e).

Option type: regole di tipo

$$\boxed{\Gamma \vdash \mathit{none} : \mathit{option}[\tau]}$$

$$\boxed{\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathit{some}(e) : \mathit{option}[\tau]}}$$

Il tipo dell'espressione esprime la possibilità di avere valori null

Ancora meglio?

- Introduzione del costruttore di tipo:

  okOrErr[τ_{ok},τ_{err}]
- per fare ``***case analysis***'' ed estrarre il valore corretto
- Un operatore del tipo:

  case e of {ok(x) ⇒ e_{ok} ; err(y) ⇒ e_{err}}
- "If e evaluates to ok(v_{ok}), then evaluate e_{ok} with v_{ok} replacing x, else it evaluates to err(v_{err}) so evaluate e_{err} with v_{err} replacing y."

Esempio: Scala

```
e match { case Ok(x) => e1
          case Err(x) => e2 }
```

I tipi Ok-errore possono essere generalizzati originando i tipi varianti.

Cosa ci rimane da fare?

- L'inferenza di tipo
- Il principio di sostituzione e il sistema dei tipi