

# Type Assignment System

Descrizione formale del sistema di tipi

- Terza fase del compilatore
- Controllo statico sul codice (albero sintattico)
  - estrarre informazioni:
    - symbol table: associa a identificatori il tipo e scope
  - eseguire i controlli non realizzabili con grammatiche liberi dal contesto
    - numero dei parametri procedura
    - identificatore dichiarato prima di essere usato
    - cicli for non modificano la variabile di ciclo
  - type checking: il controllo principale

**Testo di riferimento:** articolo di Cardelli, vedi pagina web corso:

- ripete alcune considerazioni già fatte sui sistemi di tipi
- alcune differenze nell'uso dei termini:
  - **trapped errors** errori che causano eccezioni,
  - **untrapped errors** errori che passano inosservati
  - **save** invece di **strongly typed**  
nessun untrapped error
  - **weakly checked** invece di **weakly typed**
  - **static type checking** previene alcuni trapped errors, non tutti
  - **dinamic checking** invece di **dinamic type checking**

Un sistema di tipi può essere descritto:

- informalmente nella documentazione (impreciso)
- mediante l' algoritmo di type checking, implementato come codice nel compilatore (poco comprensibile)
- mediante sistemi di regole informali

# Type system

È possibile una definizione formale del controllo di tipi

- attraverso un sistema di regole derivo **giudizi** nella forma  $x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \vdash M : A$ 
  - quando un'espressione  $M$  ha tipo  $A$ ,
  - nell'**ambiente** statico (con le ipotesi),  $x_1 : A_1, x_2 : A_2, \dots, x_n : A_n$  che assegna un tipo alle variabili in  $M$ ,
- Altri **giudizi ausiliari**
  - $x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \vdash A$   
 $A$  è un'espressione di tipo corretta nell'ambiente  $x_1 : A_1, x_2 : A_2, \dots, x_n : A_n$
  - $x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \vdash \cdot$   
l'ambiente è ben formato.
- Per i linguaggi che considereremo questi giudizi non necessari, posso definire tipi e ambienti corretti attraverso grammatiche libere

# Regole di derivazione

- simili alla deduzione naturale, o calcolo dei sequenti
- un insieme di giudizi premessa, porta ad un giudizio conclusione

$$\text{Gamma}_1 \vdash M_1 : A_1 \quad \dots \quad \text{Gamma}_N \vdash M_N : A_N$$

---

$$\text{Gamma} \vdash M : A$$

- regole senza premesse sono gli **assiomi**
- regole date per **induzione sulla struttura** di M
  - $M_1, \dots, M_n$  sottotermini di M
  - ad ogni costrutto del linguaggio si associa una regola (qualche eccezione)
- all'arricchirsi del linguaggio, aumentano le regole, approccio modulare:
  - singole regole restano valide, all'arricchirsi del linguaggio
  - regole piuttosto semplici, l'analisi del sistema di derivazione più complessa

- hanno una struttura ad albero,
  - da un insieme di assiomi: foglie
  - derivano un giudizio finale: radice
- Permettono di derivare i **giudizi validi**

Sistemi definiti mediante regole di derivazione sono frequenti:

- in logica, deduzione naturale, calcolo dei sequenti
- nella descrizione formale dei linguaggi:
  - sistema di tipi
  - semantica operativa

# Type checking, type inference

Dato un sistema di tipi, considero diversi problemi:

- problema di **controllo di tipo**, type checking,
  - dato un termine  $M$ ,
  - un tipo  $A$
  - un ambiente  $\Gamma$
  - determinare se  $\Gamma \vdash M : A$  sia valido esiste una derivazione per  $\Gamma \vdash M : A$
- problema di **inferenza di tipo**, type inference,
  - dato un termine  $M$  e un ambiente  $\Gamma$
  - trovare un tipo  $A$  tale che  $\Gamma \vdash M : A$  sia valido
- l'analisi semantica risolve problemi di type inference:
  - nel codice definito il tipo delle variabili
  - necessario inferire il tipo delle (sotto)-espressioni
  - in Python, ML, Haskell possibile non dichiarare il tipi delle variabili: necessario inferire anche l'ambiente
  - l'inferenza di tipo può essere un problema complesso



Errore di tipo, in un espressione  $M$

- non esiste  $A$  tale che  $\Gamma \vdash M : A$ 
  - definizione indiretta

Desiderata: i programmi che superano il controllo di tipo, non generano (alcuni) **errori** a tempo di esecuzione

- Per dimostrare la **type soundness** necessario collegare computazione e type system,
- possibile formulazione del collegamento:
  - **se** la computazione preserva i tipi, ossia:
    - se  $\vdash M : A$  e
    - $M$  riduce a  $N$  (viene trasformato dalla computazione in  $N$ )
    - allora  $\vdash N : A$
  - **se** il codice tipato è “ben fatto”, non contiene errori immediati
  - **allora** la valutazione di programmi tipati non genera errori

Nel seguito presentiamo, in maniera **incrementale**

- semplici linguaggi di programmazione
- relative regole di tipo

Regole illustrative:

- linguaggi diversi usano regole diverse, simili nello spirito

Regole singolarmente semplici ma sistema di tipi complesso

- numero di regole elevato
- lievi modifiche, a singole regole, possono portare ad inconsistenze

# Sistemi di tipi al prim'ordine.

- Prim'ordine, perché nessuna variabile di tipo
  - non ci sono i tipi polimorfi
- Presentiamo alcuni esempi
  - linguaggio base
  - un insieme di estensioni
  - ogni estensione richiede nuove regole
  - le regole precedenti vengono preservate  
struttura modulare.

# Sistema F1, (lambda calcolo tipato semplice)

- quasi un frammento minimo di Haskell, ma con tipi espliciti, nessun polimorfismo
- Tipi  $A, B$ 
  - un insieme di tipi base  $K$  in `Basic`
  - tipi funzioni  $A \rightarrow B$
- Termini, codice  $M, N$ 
  - variabili  $x$
  - costanti  $c$
  - funzioni  $\lambda x:A . M$
  - applicazioni  $M N$
- si associa un tipo al parametro formale di ogni funzione
  - diversamente da Haskell
  - similmente a quasi tutti gli altri linguaggi

# Regole (schemi di)

- ovvie per i giudizi di ‘buona formazione’ e ‘buon tipo’,
  - definibili mediante grammatiche libere
- (Var)

$$G, x:A, G' \vdash x:A$$

- (Fun)

$$G, x:A \vdash M:B$$

---

$$G \vdash (\lambda x:A . M) : A \rightarrow B$$

- (App)

$$G \vdash M : A \rightarrow B \quad G \vdash N:A$$

---

$$G \vdash MN : B$$

## Esempio type inference per

$\lambda f : A \rightarrow A . ( \lambda x : A . f(f x) )$

# Costruzione di una derivazione

- Costruzione top-down  
a partire da un termine da tipare
  - seleziono la regola da applicare
  - riduco il problema a quello di tipare i sottotermini
- Ad ogni passo due problemi:
  - selezionare la regola (schema di regole) da applicare  
sufficiente osservare il costrutto principale del termine
  - istanziare lo schema di regole
    - regole generiche, fanno riferimento a generici termini, tipi
    - da istanziare nel caso particolare
    - simili a meccanismi di pattern matching

Ad ogni costante si associa una regola che ne assegna tipo

Tipo base semplice, `Unit`,  
con un unico elemento, `unit`.

- `(unit)`

`G |- unit : Unit`

In Haskell: enupla vuota `() : ()`

In C, Java: `void`



Costanti - (true) (false)

$G \vdash \text{true} : \text{Bool}$        $G \vdash \text{false} : \text{Bool}$

Funzioni collegate: - (ite)

$G \vdash M : \text{Bool}$      $G \vdash N1 : A$      $G \vdash N2 : A$

-----

$G \vdash \text{if}_A M \text{ then } N1 \text{ else } N2 : A$

alternativa, meno usata:

$G \vdash (\text{if}_A \_ \text{ then } \_ \text{ else } \_) : \text{Bool} \rightarrow A \rightarrow A \rightarrow A$

Marcare il costrutto `if_A` con il tipo `A` semplifica l'inferenza di tipo, altrimenti più complessa,

## Linguaggio minimale

$G \vdash 0 : \text{Nat}$              $G \vdash \text{succ} : \text{Nat} \rightarrow \text{Nat}$

$G \vdash \text{pred} : \text{Nat} \rightarrow \text{Nat}$              $G \vdash \text{isZero} : \text{Nat} \rightarrow \text{Bool}$

i destruttori `pred`, `isZero` sostituibili con:

$G \vdash N : \text{Nat}$              $G \vdash M1 : A$              $G, x : \text{Nat} \vdash M2 : A$

---

$G \vdash \text{case\_Nat } N \text{ of } (0 \rightarrow M1) (S(x) \rightarrow M2) : A$

## Esempio, (definizione alterantiva di pred)

```
\ x:Nat . case x of (0 -> 0) (S(y) -> y)
```

# Estensioni del linguaggio dei naturali

- insieme infinito di regole per infinite costanti

$G \vdash 1 : \text{Nat}$        $G \vdash 2 : \text{Nat}$        $G \vdash 3 : \text{Nat}$     ...

- operazioni aritmetiche

$G \vdash N1 : \text{Nat}$      $G \vdash N2 : \text{Nat}$

-----

$G \vdash N1 + N2 : \text{Nat}$

regola alternativa per la somma:

$G \vdash (+) : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$

# Esempio

$\lambda y:\text{Nat} . \lambda z:\text{Nat} . (y + 2) + z$

# Tipi prodotto, coppia

- nuovo costruttore di tipi  $A * B$  in Haskell  $(A, B)$
- costruttore di elementi  $( , )$  come in Haskell

## Regole

- (Pair)

$$\begin{array}{c} G \vdash M : A \quad G \vdash N : B \\ \hline G \vdash (M, N) : A * B \end{array}$$

- (First) (Second)

$$\begin{array}{c} G \vdash M : A * B \\ \hline G \vdash \text{first } M : A \end{array} \quad \begin{array}{c} G \vdash M : A * B \\ \hline G \vdash \text{second } M : A * B \rightarrow B \end{array}$$

# Esempio

$\lambda y : (\text{Nat} * \text{Nat}) . (\text{first } y) + (\text{second } y)$

in alternativa, come in Haskell, posso usare il costrutto case (pattern matching)

- (With)

$$G \vdash M : A1 * A2 \quad G, x1:A1, x2:A2 \vdash N:B$$


---


$$G \vdash \text{case } M \text{ of } (x1:A1, x2:A2) \rightarrow N : B$$

- sintassi alternativa, in [Cardelli]

case M of (x1:A1, x2:A2) → N si scrive:

with (x1:A1, x2:A2) := M do N

- Esempio

$$\backslash y : (\text{Nat}, \text{Nat}) . \text{case } y \text{ of } (y1 : \text{Nat}, y2 : \text{Nat}) \rightarrow y1 + y2$$

Semplice generalizzare al prodotto di n tipi



Nuovo costruttore di tipi  $A + B$

Con costruttori `inLeft` in `inRight`

e regole:

- (InLeft), (InRight)

$$\frac{G \vdash M : A}{G \vdash \text{inLeft}_B M : A+B}$$
$$\frac{G \vdash M : B}{G \vdash \text{inRight}_A M : A+B}$$

# Tipi uniuone

Distruttori: `isLeft`, `isRight` e `asLeft`, `asRight`

- `(IsLeft)`, `(IsRight)`

$\frac{G \vdash M : A+B}{G \vdash \text{isLeft } M : \text{Bool}}$	$\frac{G \vdash M : A+B}{G \vdash \text{isRight } M : \text{Bool}}$
--	---

- `(AsLeft)`, `(AsRight)`

$\frac{G \vdash M : A+B}{G \vdash \text{asLeft } M : A}$	$\frac{G \vdash M : A+B}{G \vdash \text{asRight } M : B}$
--	---

l'uso `asLeft`, `asRight` può portare ad errore di tipo a tempo di esecuzione non rilevabili a tempo di compilazione

```
|- asRight (inLeft_Nat true) + 3 : Nat
```

errori rilevabili solo con type checking dinamico (dynamic checking)

È possibile evitare il type checking dinamico con il costrutto `case` simile al `case` di Haskell

$$G \vdash M : A1 + A2 \quad G, x1:A1 \vdash N1:B \quad G, x2:A2 \vdash N2:B$$


---


$$G \vdash \text{case } M \text{ of } (\text{inLeft}(x1:A1) \rightarrow N1)(\text{inRight}(x2:A2) \rightarrow N2) : B$$

- sintassi alternativa [Cardelli]

`case M of (inLeft(x1:A1)->N1)(inRight(x2:A2)->N2) : B`

scritto come:

`case M of x1:A1 then N1 | x2:A2 then N2`

# Record (Struct)

- Estensione del tipo prodotto
  - un numero arbitrario di componenti
  - sistema di etichette  $l$
- nuovo costruttore di tipo  $\{ l_1 : A_1, \dots, l_n : A_n \}$
- e di elementi  $\{ l_1 : M_1, \dots, l_n : M_n \}$

regole

- (Record)

$$\frac{G \vdash M_1 : A_1 \quad G \vdash M_2 : A_2 \quad \dots \quad G \vdash M_n : A_n}{G \vdash \{ l_1 : M_1, \dots, l_n : M_n \} : \{ l_1 : A_1, \dots, l_n : A_n \}}$$

- (Record Select)

$$\frac{G \vdash M : \{ l_1 : A_1, \dots, l_n : A_n \}}{G \vdash M.l_i : A_i}$$

Un'alternativa, poco usata, a (Record Select)

- (Record With)

$$G \vdash M : \{l_1:A_1, \dots, l_n:A_n\} \qquad G, l_1:A_1, \dots, l_n:A_n \vdash N : B$$

---

$$G \vdash \text{with } M \text{ in } N : B$$

- Estensione del tipo unione
  - un numero arbitrario di componenti
  - sistema di etichette 1

- definiscono locazioni di memoria
- introducono la memoria, lo stato
- distinguo in maniera esplicita
  - la locazione di memoria (L-value)
  - dal suo contenuto (R-value)

separazione non esplicita nei linguaggi imperativi

- costruito simile in ML
- costruttore di tipo `Ref A`
- nelle espressioni `deref` accesso al memoria, `:=` assegnazione

# Costrutti e regole

- (Ref)

$$G \vdash M : A$$

---

$$G \vdash \text{ref } M : \text{Ref } A$$

Ref M definisce una nuova locazione, inizializzata con valore M

- (Deref) accedo al contenuto

$$G \vdash \text{deref} : (\text{Ref } A) \rightarrow A$$

deref corrisponde a ! in ML, o \* in C

- (Assign)

$$G \vdash M : \text{Ref } A \quad G \vdash N : A$$

---

$$G \vdash M := N : \text{Unit}$$



- posso tradurre

```
var x = M;  
N
```

- con

```
let x = ref M in N
```

- oppure con

```
(\ x : (Ref A) . N) (ref M)
```

- posso tradurre la composizione di comandi  $C_1$ ;  $C_2$  con

```
(\ y : Unit . C2) C1
```

in un linguaggio call-by-value

In generale

- posso tradurre un linguaggio imperativo in uno funzionale + Ref

E interessante analizzare

- quali costrutti di un linguaggio posso essere derivati dagli altri
- definibili come zucchero sintattico
- regole di tipo e semantica definibili mediante traduzione

# Esempio: la regola per Composition derivabile

- (Composition)

$$G \vdash C1 : \text{Unit} \quad G \vdash C2 : \text{Unit}$$

-----

$$G \vdash C1; C2 : \text{Unit}$$
$$(C1;C2) ::= (\lambda y : \text{Unit} . C2) C1$$

## Esempio : determinare traduzione, controllare il tipo

```
var x = 3;  
    x = x + 1
```

## Esempio : determinare traduzione, controllare il tipo

```
var x = 3;
```

```
  x = x + 1
```

```
(\ x :(Ref Nat) . x := (deref x) + 1) (ref 3)
```

## Esercizi : determinare traduzione, controllare il tipo

```
var x = 3;  
    x = x + 1;  
    x = x + 2
```

## Esercizi : determinare traduzione, controllare il tipo

```
var x = 3;
```

```
  x = x + 1;
```

```
  x = x + 2
```

```
(\ x :(Ref Nat) . ((\y : Unit .   x := (deref x) + 2)  
  (x := (deref x) + 1)) (ref 3)
```

## Esercizi : determinare traduzione, controllare il tipo

```
var x = 3;  
var y = 2;  
    x = x + y
```



## Esercizi : determinare traduzione, controllare il tipo

```
var x = 3;  
var y = 2;  
    x = x + y
```

```
(\ x :(Ref Nat) .  
  (\y : (Ref Nat) .  x := (deref x) + (deref y)) (ref 2))  
  (ref 3)
```

Considero tre categorie sintattiche

- Espressioni

$$E ::= \text{const} \mid \text{id} \mid E \text{ binop } E \mid \text{unop } E$$

Considero tre categorie sintattiche

- Espressioni

$$E ::= \text{const} \mid \text{id} \mid E \text{ binop } E \mid \text{unop } E$$

- Comandi

$$C ::= \text{id} = E \mid C; C \mid \text{while } E \{C\} \mid \\ \text{if } E \text{ then } C \text{ else } C \mid I(E, \dots E) \mid \{D ; C\}$$

Considero tre categorie sintattiche

- Espressioni

$$E ::= \text{const} \mid \text{id} \mid E \text{ binop } E \mid \text{unop } E$$

- Comandi

$$C ::= \text{id} = E \mid C; C \mid \text{while } E \{C\} \mid \\ \text{if } E \text{ then } C \text{ else } C \mid I(E, \dots E) \mid \{D ; C\}$$

- Dichiarazioni

$$D ::= A \text{ id} = E \mid \text{id}(A1 \text{ id}1, \dots, A_n \text{ id}_n) \{ C \} \mid \\ \text{epsilon} \mid D; D$$

# Regole, espressioni

Per le espressioni, valgono le corrispondenti regole del linguaggio funzionale

- (Id, (Var))

$$G, id:A, G' \vdash id : A$$
$$G \vdash true : Bool \quad G \vdash false : Bool$$

- (ite)

$$G \vdash (if \_ then \_ else \_) : Bool \rightarrow A \rightarrow A \rightarrow A$$
$$G \vdash 1 : Nat \quad G \vdash 2 : Nat \quad G \vdash 3 : Nat \quad \dots$$

- operazioni aritmetiche

$$G \vdash E1 : Nat \quad G \vdash E2 : Nat$$

---

$$G \vdash E1 + E2 : Nat$$

- (Assign)

$$G \vdash \text{id} : A \quad G \vdash E : A$$

---

$$G \vdash \text{id} = E : \text{Unit}$$

- (Sequence)

$$G \vdash C1 : \text{Unit} \quad G \vdash C2 : \text{Unit}$$

---

$$G \vdash C1; C2 : \text{Unit}$$

- (While)

$$G \vdash E : \text{Bool} \quad G \vdash C : \text{Unit}$$

---

$$G \vdash \text{while } E \{C\} : \text{Unit}$$

- (If Then Else)

$$\frac{G \vdash E : \text{Bool} \quad G \vdash C1 : \text{Unit} \quad G \vdash C2 : \text{Unit}}{G \vdash \text{if } E \text{ then } C1 \text{ else } C2 : \text{Unit}}$$

- (Procedure)

$$\frac{G \vdash \text{id} : (A1 * \dots * An) \rightarrow \text{Unit} \quad G \vdash E1 : A1 \quad \dots \quad G \vdash En : An}{G \vdash \text{id } (E1, \dots, En) : \text{Unit}}$$

- (Blocco)

$$\frac{G \vdash D :: G1 \quad G, G1 \vdash C : \text{Unit}}{G \vdash \{D;C\} : \text{Unit}}$$

Le dichiarazioni necessitano di un nuovo tipo di giudizio

- Una dichiarazione crea un ambiente, che viene utilizzato nel blocco della dichiarazione
- Giudizi nella forma

$G \vdash D :: G1$

valutata nell'ambiente  $G$ ,  
la dichiarazione  $D$  crea  
i binding definiti dall'ambiente  $G1$



- (Id)

$$G \vdash E : A \quad (A \text{ tipo memorizzabile})$$

---

$$G \vdash A \text{ id} = E \quad :: \quad (\text{id} : A)$$

- (Proc)

$$G, \text{id}_1 : A_1, \dots, \text{id}_n : A_n \vdash C : \text{Unit}$$

---

$$G \vdash \text{id}(A_1 \text{id}_1, \dots \text{id}_n)\{C\} \quad :: \quad \text{id} : (A_1 * \dots * A_n) \rightarrow \text{Unit}$$

- (Recursive Proc)

$$G, \text{id}_1:A_1, \dots \text{id}_n:A_n, \text{id}:(A_1 * \dots * A_n) \rightarrow \text{Unit} \vdash C : \text{Unit}$$

---

$$G \vdash \text{id}(A_1 \text{id}_1, \dots A_n \text{id}_n)\{C\} \quad :: \quad \text{id} : (A_1 * \dots * A_n) \rightarrow \text{Unit}$$

- (Sequenza)

$$\frac{G \quad |- \quad D1 \quad :: \quad G1 \qquad G, G1 \quad |- \quad D2 \quad :: \quad G2}{G \quad |- \quad D1; D2 \quad :: \quad G1, G2}$$

# Esempio

```
Nat x = 1; update(Nat y){x = y}; while (x < 20) {update(x+2)}
```

- Devo formalizzare le due differenti interpretazioni delle espressioni
  - a sinistra della assegnazione '='
  - a destra dell'assegnazione
- finora a sinistra solo identificatori 'id'

Tipi, aggiungo un costruttore di tipo

- $A[B]$  con le opportune restrizioni
  - A a memorizzabile
  - B tipo enumerazione

Espressioni, distinguo tra espressioni sinistre e destre

- $LE ::= id \mid LE[RE]$
- $RE ::= LE \mid const \mid RE \text{ binop } RE \mid unop RE$

Una nuova versione dell'assegnamento

- $C ::= LE = RE \mid \dots$

# Regole

Per le espressioni distinguo due giudizi,

- $G \vdash_l LE : A$  (LE denota una locazione di tipo A)
- $G \vdash_r RE : A$  (RE denota un valore di tipo A)
- (Assign)

$$G \vdash_l LE : A \quad G \vdash_r RE : A$$

-----  
 $G \vdash LE = RE : \text{Unit}$

- (Left-Right)

$$G \vdash_l E : A$$

-----  
 $G \vdash_r E : A$

- (Var)

$$G, x:A, G' \vdash_l x : A$$

- (Array)

$$G \mid -l \quad E : A[B] \qquad G \mid -r \quad E1 : B$$


---


$$G \mid -l \quad E[E1] : A$$

- Dichiarazione

---


$$G \mid - \quad A[B] \text{ id} \quad :: \quad \text{id} : A[B]$$

- Le restanti regole per le espressioni restano inalterate, diventando regole per giudizi right  $\mid -r$ .

$\{ \text{Nat}[\text{Nat}] \quad V \ ; \ V[0] = 0; V[V[0]] = V[0] \}$