Type Assignment System

Descrizione formale del sistema di tipi

Analisi semantica

- Terza fase del compilatore
- Controllo statico sul codice (albero sintattico)
  - estrarre informazioni:
    - symbol table: associo a identificatori il tipo e scope
  - eseguire i controlli non realizzabili con grammatiche libere 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

Type system

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
  - safe invece di strongly typed nessun untrapped error
  - weakly checked invece di weakly typed
  - static type checking previene alcuni trapped errors, non tutti
  - dynamic checking invece di dynamic type checking

Descrizione sistema di tipi

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 di tipo nella forma $x1 : A1, x2 : A2, ...xn : An \vdash M : A$
  - quando un'espressione M ha tipo A,
  - nell'ambiente statico (con le ipotesi), $x1 : A1, x2 : A2, ...xn : An$ che assegna un tipo alle variabili in M,
- Altri giudizi ausiliari
  - $x1 : A1, x2 : A2, ...xn : An \vdash A$ A è un'espressione di tipo corretta nell'ambiente $x1 : A1, x2 : A2, ...xn : An$
  - $x1 : A1, x2 : A2, ...xn : An \vdash \cdot$ l'ambiente è ben formato.
- Per linguaggi semplici questi giudizi non sono necessari, posso definire tipi e ambienti corretti con grammatiche libere
- Notazione: uso `Gamma` o `G` per indicare un generico ambiente

Regole di derivazione

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

```
Gamma1 |- M1 : A1   ...   GammaN |- Mn : An
-----------------------------------------
                Gamma |- M : A
```

- regole senza premesse sono gli assiomi
- regole date per induzione sulla struttura di `M`
  - `M1`, `...` , `Mn` 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

Derivazioni

- hanno una struttura ad albero,
  - da un insieme di assiomi: foglie
  - derivo 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 operazionale

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

Type soundness

Errore di tipo, in un'espressione `M`

- non esiste `A` tale che `Gamma |- 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 `|- M : A`
    - `M` riduce a `N` (viene trasformato dalla computazione in `N`)
    - allora `|- N : A`
  - se il codice tipato è "ben fatto", non contiene errori immediati
  - allora la valutazione di programmi tipati non genera errori

Esempi sistemi di tipi

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 a 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 -> B`
- Termini, codice `M, N`
  - variabili `x`
  - costanti `c`
  - funzioni `\ 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' |- x:A
```

- (Fun)

```
     G, x:A |- M:B
------------------------
G |- (\ x:A . M) : A->B
```

- (App)

```
G |- M : A->B      G |- N:A
--------------------------
    G |-  MN :  B
```

Esempio type inference per

```
\ f : A->A . ( \ 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 e tipi
    - da istanziare nel caso particolare
    - simili a meccanismi di pattern matching

Tipi base e costanti

Ad ogni costante si associa una regola che ne assegna il tipo

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

- (unit)

```
G |- unit : Unit
```

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

In C, Java: `void`

Booleani

Costanti - (true) (false)

```
G |- true : Bool      G |- false : Bool
```

Funzioni collegate: - (ite)

```
G |- M : Bool   G |- N1 : A  G |- N2 : A
-------------------------------------
   G |- if_A M then N1 else N2 : A
```

alternativa, meno usata:

```
 G |- (if_A _ then _ else _) : Bool -> A -> A -> A
```

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

Naturali

Linguaggio minimale

```
G |- 0 : Nat        G |- succ : Nat -> Nat

G |- pred : Nat -> Nat        G |- isZero : Nat -> Bool
```

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

```
G |- N:Nat     G |- M1:A     G, x:Nat |- M2:A
-----------------------------------------------
G |- case_Nat N of (0 -> M1) (S(x) -> M2) : A
```

Esempio, (definizione alternativa 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 |- 1 : Nat      G |- 2 : Nat      G |- 3 : Nat  ...
```

- operazioni aritmetiche

```
G |- N1 : Nat  G |- N2 : Nat
---------------------------
   G |- N1 + N2 : Nat
```

regola alternativa per la somma:

```
G |- (+) : Nat -> Nat -> Nat
```

Esempio

```
\ y:Nat . \ z: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)

```
G |- M : A     G |- N : B
-------------------------
    G |- (M,N) : A*B
```

- (First) (Second)

```
  G |- M : A*B          G |- M : A*B
---------------      ----------------------
G |- first M : A     G |- second M : A*B -> B
```

Esempio

```
\ y : (Nat*Nat) . (first y) + (second y)
```

segue

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

- (Case)

```
G |- M : A1 * A2      G, x1:A1, x2:A2 |- N:B
-----------------------------------------
G |- case M of (x1:A1,x2:A2) -> 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

```
\ y : Nat*Nat . case y of (y1 : Nat, y2: Nat) -> y1 + y2
```

Semplice generalizzare alle enuble (prodotti di n tipi)

Tipi unione

Nuovo costruttore di tipi `A + B`

Con costruttori di elementi `inLeft` e `inRight`

e regole:

- (InLeft), (InRight)

```
   G |- M : A                      G |- M : B
--------------------      ----------------------
G |- inLeft_B M : A+B     G |- inRight_A M : A+B
```

Tipi unione

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

- (AsLeft), (AsRight)

```
   G |- M : A+B                    G |- M : A+B
-----------------              -----------------
G |- asLeft M : A              G |- asRight M : B
```

- (IsLeft), (IsRight)

```
   G |- M : A+B                      G |- M : A+B
-----------------                -----------------
G |- isLeft M : Bool             G |- isRight M : Bool
```

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)

segue

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

```
G |- M : A1 + A2     G, x1:A1 |- N1:B     G, x2:A2 |- N2:B
-------------------------------------------------------------
G |- case M of(inLeft(x1:A1) -> N1)(inRight(x2:A2) -> 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 `{ l1 : A1, ... , ln : An }`
- e di elementi `{ l1 : M1, ... , ln : Mn }`

regole

- (Record)

```
G |- M1 : A1      G |- M2:A2     ...     G |- Mn:An
-----------------------------------------------------
G |- { l1:M1, ..., ln:Mn } : { l1:A1, ..., ln:An }
```

- (Record Select)

```
G |- M : { l1 : A1, ... , ln : An }
-----------------------------------
        G |-  M.li : Ai
```

segue

Un'alternativa, poco usata: Pascal - JavaScript, a (Record Select)

- (Record With)

```
G |- M:{l1:A1, ..., ln:An}      G, l1:A1,... , ln:An |- N : B
-------------------------------------------------------------
     G |- with M in N : B
```

Variant type

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

Reference type

- 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
- costrutto simile in ML
- costruttore di tipo `Ref A`
- per creare locazioni `ref`
- nelle espressioni:
  - `deref` accesso alla locazione,
  - `:=` assegnazione

Costrutti e regole

- (ref)

```
G |- M : A    A memorizabile
-----------------------------
     G |- ref M : Ref A
```

`ref M` definisce una nuova locazione, inizializzata con valore `M`

- (deref) accedo al contenuto della locazione

```
G |- deref : (Ref A) -> A
```

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

- (Assign)

```
G |- M : Ref A      G |- N : A
-------------------------------
    G |- M := N  :  Unit
```

Linguaggio imperativo dentro uno funzionale

- posso tradurre

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

- con

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

- oppure, in un linguaggio senza `let` con

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

- posso tradurre la composizione di comandi `C1; C2` con

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

in un linguaggio call-by-value

Linguaggio imperativo dentro uno funzionale

In generale

- posso tradurre un linguaggio imperativo in uno funzionale + Ref

E interessante analizzare

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

Esempio: la regola per Composition derivabile

- (Composition)

```
G |- C1 : Unit      G |- C2 : Unit
-------------------------------
   G |- C1; C2  :  Unit

(C1;C2)  ::=  (\ y : 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)
```

Linguaggio imperativo, simil C.

Considero tre categorie sintattiche

- Espressioni

```
E ::= const  |  id  |  E binop E  |  unop E
```

Linguaggio imperativo, simil C.

Considero tre categorie sintattiche

- Espressioni

```
E ::= const  |  id  |  E binop E  |  unop E
```

- Comandi

```
C ::= id = E  |  C; C  |  while E {C}  |
            if E then C else C  |  I(E, ..., E)  |  {D ; C}
```

Linguaggio imperativo, simil C.

Considero tre categorie sintattiche

- Espressioni

```
E ::= const  |  id  |  E binop E  |  unop E
```

- Comandi

```
C ::= id = E  |  C; C  |  while E {C}  |
            if E then C else C  |   I(E, ..., E)  |   {D ; C}
```

- Dichiarazioni

```
D  ::=  A id = E  |  id(A1 id1, ... , An idn) { C }  |
            epsilon  |  D; D
```

Regole, espressioni

Per le espressione, valgono le corrispondenti regole del linguaggio funzionale

- (Id, (Var))

```
G, id:A, G' |- id : A
```

```
G |- true : Bool      G |- false : Bool
```

- (ite)

```
 G |- (if _ then _ else _) : Bool -> A -> A -> A
```

```
G |- 1 : Nat       G |- 2 : Nat      G |- 3 : Nat   ...
```

- operazioni aritmetiche

```
G |-  E1 : Nat      G |- E2 : Nat
------------------------------
    G |-  E1 + E2 : Nat
```

Comandi

- (Assign)

```
G |- id : A      G |- E : A
--------------------------
   G |- id = E  :  Unit
```

- (Sequence)

```
G |- C1 : Unit  G |- C2 : Unit
------------------------------
  G |- C1; C2  :  Unit
```

- (While)

```
G |- E : Bool      G |- C : Unit
--------------------------------
   G |- while E {C}  :  Unit
```

segue

- (If Then Else)

```
G |- E : Bool     G |- C1 : Unit  G |- C2 : Unit
-----------------------------------------------
      G |- if E then C1 else C2  :  Unit
```

- (Procedure)

```
G |- id:(A1 * ... * An)->Unit  G |- E1 : A1 ...  G |- En : An
-----------------------------------------------------------
          G |- id (E1, ..., En) : Unit
```

- (Blocco)

```
G |- D :: G1     G, G1 |- C : Unit
---------------------_----------
       G |- {D;C} : Unit
```

Dichiarazioni

Le dichiarazione necessitano di un nuovo tipo di giudizio

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

```
G |- D :: G1
```

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

Regole

- (Id)

```
G |-  E : A     (A tipo memorizzabile)
-------------------------------------
    G |-  A id = E  ::  (id : A)
```

- (Proc)

```
G, id1 : A1, ..., idn : An |-  C : Unit
---------------------------------------------------
G |- id(A1 id1, ... idn){C} :: id : (A1 *...*  An) -> Unit
```

- (Recursive Proc)

```
G, id1:A1, ... idn:An, id:(A1 *...* An)->Unit  |-  C : Unit
-----------------------------------------------------------
G |- id(A1 id1, ... An idn){C} :: id : (A1*...* An)->Unit
```

segue

- (Sequenza)

```
G  |-  D1 :: G1        G, G1 |-  D2 :: G2
-----------------------------------------
       G |-  D1; D2 :: G1, G2
```

Esempio

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

Array

- 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 | LE[RE]`
- `RE ::= LE | const | RE binop RE | unop RE`

Una nuova versione dell’assegnamento

- `C ::= LE = RE | ...`

Regole

Per le espressioni distinguo due giudizi,

- `G |-l LE : A` (`LE` denota una locazione di tipo `A`)
- `G |-r RE : A` (`RE` denota un valore di tipo `A`)
- (Assign)

```
G |-l  LE : A      G |-r  RE : A
-------------------------------
G |-   LE = RE : Unit
```

- (Left-Right)

```
G |-l  E : A
------------
G |-r  E : A
```

- (Var)

```
G, x:A, G' |-l x : A
```

segue

- (Array)

```
G |-l  E : A[B]      G |-r  E1 : B
-------------------------------
     G |-l  E[E1] : A
```

- Dichiarazione

```
-----------------------------
G |-  A[B] id   ::  id : A[B]
```

- Le restanti regole per le espressioni restano inalterate, diventando regole per giudizi right `|-r`.

Esempi

```
{ Nat[Nat]  V;  V[O] = O; V[V[O]] = V[O]}
```

Polimorfismo di sottotipo.

Maggiore flessibilità permetto ad un espressione di avere più tipi

- Varie forme di polimorfismo
  - ad hoc
  - di sottotipo
  - parametrico
  - combinazioni dei precedenti
- Sottotipo (e ad hoc)
  - introduco una relazione di sottotipo, con relativi giudizi e regole

```
G |- A <: B
```

- (Subsumtion)

```
G |-r  E : A     G |-  A <: B
-----------------------------
       G |-r  E : B
```

Regole per i giudizi di sottotipo

Posso trattare il polimorfismo ad-hoc, con assiomi del tipo.

```
G |-  Int <: Float
```

Polimorfismo dei linguaggi ad oggetti:

- un tipo oggetto con più campi, sottoggetto di uno con meno.
- formalizziamolo con i tipi Record

```
G |- A1 <: B1 ...      G |- Am <:Bm      m < n
-----------------------------------------------
G |- { l1:A1, ..., ln:An } <: { l1:B1, ..., lm:Bm}
```

segue

Definisco regole di sottotipo per ogni costruttore di tipi.

- (Prod <:)

```
G |- A1 <: B1       G |- A2 <:B2
-------------------------------
G |- A1 * A2  <:  B1 * B2
```

Regole sempre covariante con eccezione

- (Arrow <:)

```
G |- A1 <: B1       G |- A2 <:B2
-------------------------------
G |- B1 -> A2  <:  A1 -> B2
```

`controvariante sul primo argomento.`