

Type checking

Lunedì 7 Gennaio

Type checking

(Controllo dei tipi)

- La TS, o meglio il suo contenuto, è molto importante per uno dei principali compiti della fase di analisi semantica: il ***type checking***.
- In effetti, con “Type checking” si fa riferimento a due attività separate ma strettamente correlate fra di loro (eseguite insieme):
 - calcolo e mantenimento delle informazioni sui tipi dei dati;
 - controllo che ogni parte del programma abbia un senso per le regole dei tipi ammessi nel linguaggio.

Type checking

(Controllo dei tipi)

- Le informazioni sui tipi dei dati possono essere **statiche** o **dinamiche**.
- In alcuni linguaggi (LISP) la gestione dei tipi dei dati è interamente dinamica. In questo caso il **Type checking** verrà eseguito durante l'esecuzione.
- In altri linguaggi (C, Pascal) le informazioni sono prevalentemente statiche. In questo caso il **Type checking** (type inference + type checking) verrà eseguito durante il processo di compilazione.
- Le informazioni statiche sono anche utili per determinare lo spazio di memoria necessario per allocare le variabili.
- Ovviamente noi ci occuperemo solo del **Type checking** statico.

Compatibilità o equivalenza di tipi

- Il Type checker controlla che in un'espressione gli operandi siano compatibili fra di loro e, se necessario, che il risultato sia compatibile con la variabile destinata a riceverlo.
- Domanda: “quando due operandi sono compatibili?”
- Non basta dire: “quando sono dello tipo”.
- Infatti l'introduzione delle dichiarazioni di tipo nei moderni linguaggi di programmazione pone una serie di questioni che il progettista di compilatori (e non solo) deve tenere in considerazione.

Compatibilità o equivalenza di tipi

- La risposta sulla compatibilità dei tipi dipende infatti dalla nozione di compatibilità o equivalenza fra tipi che viene adottata.
- Si considerano due diverse nozioni di equivalenza:
 - Equivalenza nominale
 - Equivalenza strutturale

Equivalenza di tipi

- **Equivalenza nominale**: secondo questa nozione due variabili sono dello stesso tipo se e solo se appaiono nella stessa lista di una dichiarazione oppure se sono state dichiarate con lo stesso **<nome-tipo>** (sia esso predefinito o definito dall'utente).
- **Equivalenza strutturale**: secondo questa nozione due variabili sono dello stesso tipo se hanno la stessa struttura.

Equivalenza di tipi

- **L'equivalenza nominale** risulta molto più restrittiva dell'**equivalenza strutturale**: due tipi equivalenti nominalmente lo sono anche strutturalmente ma non è generalmente vero il contrario.
- Solo da alcuni anni, per evitare differenti interpretazioni da parte dei compilatori si è ritenuto essenziale includere nelle descrizione di un linguaggio anche le regole di equivalenza dei tipi.

Dichiarazioni

- Supponiamo di poter dichiarare una sola variabile per volta.
- Una grammatica semplificata potrebbe essere la seguente:
D \rightarrow T id; D | ϵ
T \rightarrow B C | record '{' D '}'
B \rightarrow int | float
C \rightarrow ϵ | '['num']' C
- Il tipo e l'indirizzo relativo di un certo identificatore sono memorizzati nella ST.

Regole per il type checking

- Il type checking può agire secondo due metodologie:
 - Sintesi: costruzione di un tipo di un'espressione a partire dal tipo delle sue sottoespressioni. Richiede che tutti i nomi siano stati dichiarati prima.
 - Inferenza: determina il tipo di un costrutto di un linguaggio in base al modo in cui è usato.

Realizzazione di un type checker

- Faremo adesso un esempio di un semplice linguaggio per il quale descriveremo il Type checking in termini di azioni semantiche.
- Introduurremo i due attributi:
 - name
 - type
- Faremo uso della TS.

Le regole della grammatica

$program \rightarrow var-decls ; stmts$
 $var-decls \rightarrow var-decls ; var-decl \mid var-decl$
 $var-decl \rightarrow \mathbf{id} : type-exp$
 $type-exp \rightarrow \mathbf{int} \mid \mathbf{bool}$
 $stmts \rightarrow stmts ; stmt \mid stmt$
 $stmt \rightarrow \mathbf{if} \ exp \ \mathbf{then} \ stmt \mid \mathbf{id} := exp$
 $exp \rightarrow exp + exp$
 $exp \rightarrow exp \ \mathbf{or} \ exp$
 $exp \rightarrow \mathbf{true}$
 $exp \rightarrow \mathbf{false}$
 $exp \rightarrow \mathbf{id}$

Grammar Rule	Semantic Rules
$var-decl \rightarrow id : type-exp$	$insert(id.name, type-exp.type)$
$type-exp \rightarrow int$	$type-exp.type := integer$
$type-exp \rightarrow bool$	$type-exp.type := boolean$
$stmt \rightarrow if\ exp\ then\ stmt$	$if\ not\ typeEqual(exp.type, boolean)$ $\quad then\ type-error(stmt)$
$stmt \rightarrow id := exp$	$if\ not\ typeEqual(lookup(id.name),$ $\quad exp.type)\ then\ type-error(stmt)$
$exp_1 \rightarrow exp_2 + exp_3$	$if\ not\ (typeEqual(exp_2.type, integer)$ $\quad and\ typeEqual(exp_3.type, integer))$ $then\ type-error(exp_1) ;$ $exp_1.type := integer$
$exp_1 \rightarrow exp_2\ or\ exp_3$	$if\ not\ (typeEqual(exp_2.type, boolean)$ $\quad and\ typeEqual(exp_3.type, boolean))$ $then\ type-error(exp_1) ;$ $exp_1.type := boolean$
$exp \rightarrow true$	$exp.type := boolean$
$exp \rightarrow false$	$exp.type := boolean$
$exp \rightarrow id$	$exp.type := lookup(id.name)$

Type checking delle dichiarazioni

$var\text{-}decl \rightarrow \mathbf{id} : type\text{-}exp$	$insert(\mathbf{id}.name, type\text{-}exp.type)$
$type\text{-}exp \rightarrow \mathbf{int}$	$type\text{-}exp.type := integer$
$type\text{-}exp \rightarrow \mathbf{bool}$	$type\text{-}exp.type := boolean$

- La regola semantica

$insert(\mathbf{id}.name, type\text{-}exp.type)$

associa all'identificatore inserito nella TS il suo tipo.

Type checking degli statement

- Gli statement non hanno un loro tipo ma bisogna sempre verificarne la correttezza.

$stmt \rightarrow \mathbf{if} \ exp \ \mathbf{then} \ stmt$	$\mathbf{if} \ \mathbf{not} \ typeEqual(exp.type, boolean)$
	$\mathbf{then} \ type-error(stmt)$

- Nel caso del costrutto **if** si richiede che l'espressione condizionale sia di tipo boolean.
- La funzione **typeEqual** stabilisce se i tipi rappresentati dai suoi due parametri sono equivalenti (per nome).
- La funzione **type-error** segnalerà opportunamente un errore in funzione dello statement esaminato.

Type checking degli statement

- Nel caso dell'assegnazione si richiede che il risultato dell'espressione sia di tipo compatibile con quello della variabile assegnata.

$stmt \rightarrow id := exp$	if not $typeEqual(lookup(id.name),$ $exp.type)$ then $type-error(stmt)$
------------------------------	--

- La funzione **lookup** restituisce il tipo associato all'identificatore passato come parametro.

Type checking delle espressioni

$exp_1 \rightarrow exp_2 + exp_3$	if not (<i>typeEqual</i> ($exp_2.type$, <i>integer</i>) and <i>typeEqual</i> ($exp_3.type$, <i>integer</i>)) then <i>type-error</i> (exp_1) ; $exp_1.type := integer$
-----------------------------------	--

$exp_1 \rightarrow exp_2 \text{ or } exp_3$	if not (<i>typeEqual</i> ($exp_2.type$, <i>boolean</i>) and <i>typeEqual</i> ($exp_3.type$, <i>boolean</i>)) then <i>type-error</i> (exp_1) ; $exp_1.type := boolean$
---	--

$exp \rightarrow \text{true}$	$exp.type := boolean$
-------------------------------	-----------------------

$exp \rightarrow \text{false}$	$exp.type := boolean$
--------------------------------	-----------------------

$exp \rightarrow id$	$exp.type := lookup(id.name)$
----------------------	-------------------------------

Type coercion

(conversione automatica di tipo)


- In alcuni linguaggi di programmazione è possibile che alcune espressioni, che coinvolgono operandi di tipo diverso, vengano valutate correttamente dopo opportune conversioni (ogni linguaggio specifica opportune regole di conversione).
- Le conversioni possono riguardare sia le espressioni che le assegnazioni.
- Le conversioni possono essere:
 - implicite: effettuate dal type checker automaticamente.
 - esplicite: è il programmatore che forza la conversione (cast).

Type coercion delle espressioni

- Consideriamo le seguenti regole:
 - $\text{exp}_1 \rightarrow \text{exp}_2 + \text{exp}_3$
 - $\text{type-exp} \rightarrow \text{int}$
 - $\text{type-exp} \rightarrow \text{real}$
- L'azione semantica che esegue la conversione di tipo può essere:

```
if typeEqual(exp2.type, integer) and typeEqual(exp3.type, integer) then exp1.type:=integer
else
if typeEqual(exp2.type, integer) and typeEqual(exp3.type, real) then exp1.type:=real
else
if typeEqual(exp2.type, real) and typeEqual(exp3.type, integer) then exp1.type:=real
else
if typeEqual(exp2.type, real) and typeEqual(exp3.type, real) then exp1.type:=real
else type-error(exp1)
```

Type coercion delle espressioni

- Consideriamo le seguenti regole:
 - $\text{exp}_1 \rightarrow \text{exp}_2 + \text{exp}_3$
 - $\text{type-exp} \rightarrow \text{int}$
 - $\text{type-exp} \rightarrow \text{real}$
- Usa due funzioni:
 - $\text{Max}(t1, t2)$ restituisce il maggiore dei due tipi $t1$ e $t2$ secondo la gerarchia di promozione.


Byte — Short — int — long — float — double
char — int
 - $\text{Widen}(a, t, w)$ genera la conversione di tipo necessaria per promuovere un indirizzo di tipo t in uno di tipo w

Overloading

- Un operatore (o una funzione) è overloaded se lo stesso nome di operatore ha diverso significato a seconda del contesto.

Es.1 `2+3` (somma tra interi)
 `2.1 +3.0` (somma tra reali)

Es.2 `int max(int x,y);`
 `float max (float x,y);`

Un modo per eliminare l'ambiguità consiste nell'effettuare le operazioni di lookup nella TS controllando anche la lista dei parametri.

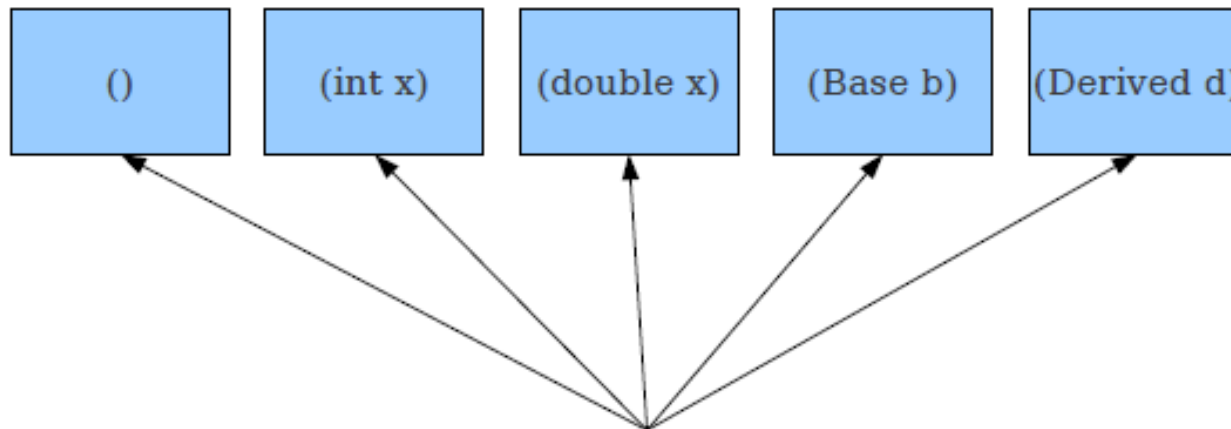
Esempio

```
void Function() ;  
void Function(int x) ;  
void Function(double x) ;  
void Function(Base b) ;  
void Function(Derived d) ;
```

```
Function() ;  
Function(137) ;  
Function(42.0) ;  
Function(new Base) ;  
Function(new Derived) ;
```

Come si realizza?

```
void Function();  
void Function(int x);  
void Function(double x);  
void Function(Base b);  
void Function(Derived d);
```



Function(137);

- Si verifica con quale lista di parametri formali, i parametri attuali hanno un match.

Polimorfismo

- Una funzione è polimorfa se può avere parametri di qualsiasi tipo.

```
Procedure swap(var x,y:anytype);
```

A differenza dell'overloading in cui ci sono funzioni diverse con lo stesso nome, nel polimorfismo una stessa funzione può essere applicata a variabili di più tipi.

Esistono particolari type checker che, sfruttando particolari e sofisticate tecniche di pattern matching risolvono il problema.

Equivalenza strutturale

- L'unificazione è il problema di determinare se due espressioni s e t possono essere rese identiche sostituendo alle variabili di s e t nuove espressioni.
- Si risolve attraverso algoritmi su grafi.
- La verifica dell'equivalenza strutturale è un particolare problema di unificazione.