Type Checking (1° parte)

Implementazione dei linguaggi 2

Simone Vallarino

Testo:

A.V. Aho, R. Sethi, J.D.Ullman

Compilers Principles,Techniques and Tools, Addison Wesley

Sommario

- Type checking.
- Type system.
  - Type espression.
  - Type system.
- Esempio di un semplice type checker.
  - Il linguaggio.
  - Schema di traduzione.
  - Type checking di espressioni.
  - Type checking di istruzioni.
  - Type checking di funzioni.
- Equivalenza di tipo.

Type checking

- Un compilatore deve controllare che il programma sorgente segua sia le convenzioni sintattiche che quelle semantiche del linguaggio sorgente.

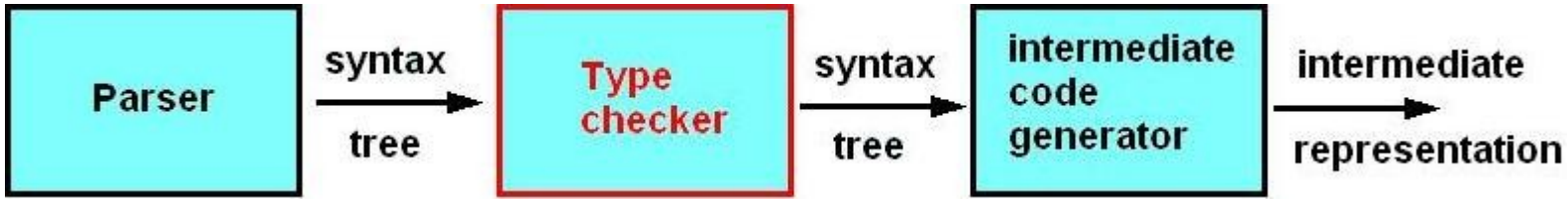
- Si divide in:
  - statico se eseguito durante la compilazione.
  - dinamico se eseguito durante l'esecuzione del programma.

Type checking

- È affiancato da altri controlli di tipo statico:
  - *Flow of control checks*: le istruzioni che causano al controllo di flusso di lasciare un costrutto devono avere dello spazio per trasferirlo. (goto, return).
  - *Controlli di unicità*: ci sono casi in cui occorre che un oggetto sia definito una sola volta (identificatori nello stesso scope, label dei switch/case).
  - *Controlli contestuali sui nomi*: alcune volte lo stesso nome deve apparire più volte.
    - es. (Modula) PROCEDURE P; … END P;

Type checking

- Il type checking controlla che le operazioni del linguaggio vengano applicate ad operandi di tipo compatibile.
- Il progetto di un type checker per un linguaggio è basato sulle informazioni circa i costrutti sintattici, la nozione di tipi, e le regole per assegnare tipi ai costrutti del linguaggio.



Type expression

- Denotano i tipi di un linguaggio e sono quindi fortemente dipendenti da esso.
- Una type expression è un tipo base o è formata applicando un operatore type constructor (costruttore di tipo) ad altre type expression.
- L'insieme dei tipi base e dei costruttori dipende dal linguaggio che deve essere controllato.

Type expression

- Definizioni di Type Expression:
  - Un tipo base (int,float, integer, real, complex...) è una type expression.

    + type_error per gestire errori di tipo.

    + void per indicare assenza di valore.

  - Un tipo può avere un nome, per cui il nome di un tipo è una type expression.

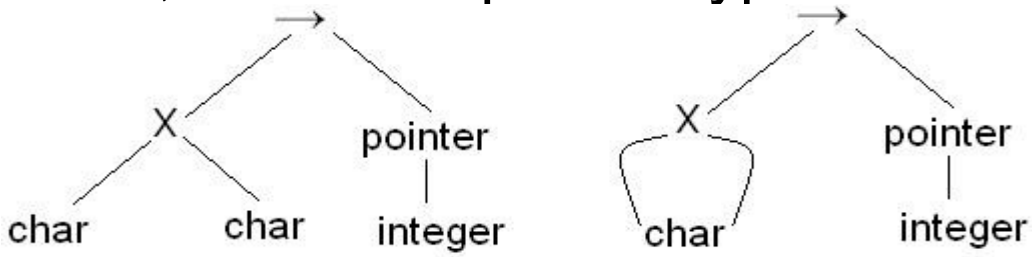
  - Un costruttore di tipo applicato ad una type expression produce una type expression.

Type expression

- Ci sono vari casi:
  - *Array*: Se T è una T.E. allora array (I,T) è una T.E. che denota il tipo di un array di tipo base T e insieme di indici (tipo indice) I.
  - *Prodotti cartesiani*: se T_1 e T_2 sono T.E. allora $T_1 x T_2$ è una T.E.(denota l'ins delle coppie (t_1,t_2) con $t_1 \in T_1$ e $t_2 \in T_2$) (x associa a sx).
  - *Record*: è il prodotto del tipo dei suoi campi. Differisce dal prodotto per la presenza di un nome che identifica i campi (elem commutativi grazie a selezione mediante nome).
  - *Puntatori*: se T è una T.E. allora pointer(T) è una T.E. che denota il tipo puntatore ad un oggetto di tipo T.
  - *Funzioni*: se una funzione mappa un domain type D in un range type R allora il tipo della funzione è denotato dalla T.E. D→R.
  - *Class* : è un incrocio tra il concetto di modulo e di tipo. È modellabile come un record con campi di tipo procedura e funzione detti metodi.

Type expression

- Le type expression possono contenere variabili il cui valore è una T.E..
- Quest'ultime sono dette Type variables.
- Un modo conveniente per rappresentare type expression è usare un grafo.
  - Si può costruire un tree o un dag per una T.E. con i nodi interni per i type constructors e le foglie per i tipi base, i nomi di tipo e le type variables.



Type system

- Collezione di regole per assegnare T.E. alle varie parti di un programma.
- Un Type system è implementato da un type checker.
- Diversi type system possono essere usati da diversi compilatori e processori dello stesso linguaggio.
- Il controllo eseguito da un compilatore è detto statico mentre quello eseguito mentre il target program è in esecuzione è detto dinamico.
- Ogni tipe checker può essere applicato a run-time se il codice oggetto contiene informazioni che specificano il tipo degli elementi.

Type system

- T.S. sound: elimina la necessità di type checking dinamico (se un T.S. assegna un tipo != type_error ad una parte di programma allora non ci saranno errori di tipo nell'esecuzione del codice oggetto corrispondente).

- Un linguaggio è fortemente tipato se il suo compilatore può garantire che il programma che "accetta"verrà eseguito senza errori di tipo.

Type system

- Error recovery:
- il T.C. può catturare gli errori nei programmi ma è anche importante fare qualcosa di ragionevole quando viene scoperto un errore:
  - Il compilatore deve almeno riportare la natura e la posizione dell'errore.
  - Sarebbe utile recuperare dagli errori in modo da continuare con il controllo del resto dell'input.
- La gestione degli errori modifica le regole di type checking quindi deve essere progettata nel type system fin dall'inizio.

Un semplice type checker

- Il tipo di ogni identificatore deve essere dichiarato prima che l'identificatore stesso venga usato.
- Il T.C. è uno schema di traduzione che sintetizza il tipo di ogni espressione dal tipo delle sue sottoespressioni.
- Il T.C. può gestire: array, puntatori, istruzioni e funzioni.

Il linguaggio

- Un esempio:
- La seguente grammatica genera programmi (P) che consistono di una sequenza di dichiarazioni (D) seguite da espressioni singole (E).

  P→D;E

  D → D;D | id: T

  T →char | integer | array[num] of T| ↑T

  E →literal | num | id | E mod E | E[E] | E↑
- Esempio

  key: integer;

  key mod 1999.

Il linguaggio

- Il linguaggio ha due tipi base: char e integer.
- Un terzo tipo base è il type_error usato per segnalare e gestire gli errori di tipo.
- Assumiamo che gli array partano da 1 e che ↑ si riferisca ad un tipo puntatore.
  - Array[256] of char → array(1..256, char)
  - ↑integer → pointer(integer)

Schema di traduzione

$P \rightarrow D;E$

$D \rightarrow D;D$

$D \rightarrow id:T \quad \{addtype(id.entry, T.type)\}$

$T \rightarrow char \quad \{T.type:=char\}$

$T \rightarrow integer \quad \{T.type:=integer\}$

$T \rightarrow array[num]\ of\ T_1 \quad \{T.type:=array(num.val, T_1)\}$

$T \rightarrow \uparrow T_1 \quad \{T.type:=pointer(T_1.type)\}$

Schema di traduzione

- La produzione associata a D→ id: T salva un tipo in una entry della symbol table per un identificatore.
  - Azione: {addtype(id.entry,T.type)}
- Da notare che dato che D appare prima di E (P → D; E) siamo sicuri che tutti I tipi degli identificatori dichiarati sono stati salvati prima che l'espressione generata da E sia controllata.

Type checking di espressioni

- In queste regole l'attributo di tipo per E dà la T.E. assegnata dal type system all'espressione generata da E.

- Le costanti rappresentate dai tokens literal e num hanno tipo char e integer rispettivamente.

  $E \rightarrow$ literal $\quad \{E.type := char\}$

  $E \rightarrow$ num $\quad \{E.type := integer\}$

- Usiamo una funzione lookup(e) per trovare il tipo salvato nella symbol table. Quando in una E appare un id il suo tipo dichiarato è trovato e assegnato all'attributo di tipo.

  $E \rightarrow$ id $\quad \{E.type := lookup(id.entry)\}$

Type checking di espressioni

- Operatore Mod:

  $E \rightarrow E_1$ mod E_2 {E.type := if (E1.type=integer & E2.type=integer) then integer else type_error}

- Array:

  $E \rightarrow E_1[E_2]$ {E.type := if (E_2.type=integer & E_1.type=array(s,t)) then t else type_error}

- Puntatori:

  $E \rightarrow E_1 \uparrow$ {E.type := if E_1.type=pointer(t) then t else type_error}

Type checking di espressioni

- Volendo si potrebbe aggiungere qualcosa alla grammatica:
  - Ad esempio
    T → boolean

    Si permetterebbe agli id di avere tipo boolean, e con l'introduzione di operatori di confronto (es. "<") e connettivi logici (es. "and") nelle produzioni per E si permetterebbe la costruzione di espressioni di tipo booleano.

Type checking di istruzioni

- Le istruzioni solitamente non hanno valore: si assegna allora il tipo base void. Se c'è un errore si usa type_error.
- Consideriamo: assegnamenti, condizioni e while.

$S \rightarrow$ id := E $\quad \{$S.type:=if id.type=E.type then void else type_error$\}$

Controlla che il lato dx e sx di un assegnamento abbiano lo stesso tipo.

Type checking di istruzioni

$S \rightarrow$ if E then S_1 {S.type:= if E.type =boolean then S_1.type else type_error}

$S \rightarrow$ while E do S_1 {S.type:=if E.type=boolean then S_1.type else type_error}

Specificano che espressioni in istruzioni if e while devono avere tipo booleano.

$S \rightarrow S_1;S_2$ {S.type:=if S_1.type=void & S2.type=void then S.type=void else type_error}

Una sequenza di istruzioni ha tipo void solo se ogni sottoistruzione ha tipo void. Un type mismatch produce type_error.

Type checking di istruzioni

- Queste produzioni possono essere combinate con quelle della grammatica viste in precedenza cambiando la produzione per un programma completo da P→D;E a P→D;S.

- Ora Programma= dichiarazioni seguite da istruzioni ma le regole per il controllo delle espressioni sono sempre necessarie (le istruzioni possono avere espressioni al loro interno).

Type checking di funzioni

- L'applicazione di una funzione ad un argomento può essere "catturata" dalla produzione.
  - E→ E(E)
- La regola che associa T.E. con non terminali T può essere espansa per permettere tipi funzione nelle dichiarazioni ('→' = costruttore di funzioni).
  - T → T1 '→' T2 {T.type := T1.type → T2.type}
- La regola per controllare il tipo di una funzione è:
  - $E \rightarrow E_1(E_2)$ {E.type:=if E_2.type=s & E1.type=s→t then t else type_error}
- Funzioni con più di un parametro si risolvono creando un tipo prodotto con i vari parametri (T1 x T2 x … xTn).

Equivalenza di type expression

- Le regole che abbiamo visto erano del tipo: se due T.E. sono uguali ritorna un certo tipo altrimenti ritorna errore.
- Ma quando due T.E. sono uguali?.
- Equivalenza strutturale:

  due T.E.sono equivalenti sse sono identiche.

Equivalenza di tipo

- Tipi riferiti in due o più parti di un programma sono:
- Identici:
  - se in tali parti viene usato lo stesso identificatore di tipo o
  - se ne vengono usati diversi (T1 e T2) definiti equivalenti da una definizione di tipo T1=T2.
- Compatibili:
  - se sono identici o
  - se uno è sottorango dell'altro o
  - se entrambi sono sottorango dello stesso tipo o
  - se sono di tipo stringa con lo stesso n di componenti o
  - Se sono tipi set di tipo base compatibile.

Equivalenza di tipo

- Un’espressione E di tipo T2 è detta compatibile rispetto all’assegnazione con un tipo T1 se una delle seguenti proposizioni è vera:
  - T1 e T2 sono identici e nessuno dei due è di tipo file o un tipo strutturato con una componente di tipo file.
  - T1 è di tipo real e T2 è di tipo integer.
  - T1 e T2 sono di tipo ordinale compatibile e il valore di E. appartiene all’intervallo chiuso specificato dal tipo di T1.
  - T1 e T2 sono tipi set compatibili e tutti i membri del valore del set E appartengono all’intervallo chiuso specificato dal tipo base di T1.
  - T1 e T2 sono tipi stringa compatibili.

Equivalenza di tipo

- Dove viene applicata la regola di compatibilità relativa all'assegnazione si ha:
  - Se T1 e T2 sono tipi ordinali compatibili e il valore dell'espressione E non appartiene all'intervallo chiuso specificato dal tipo T1 si ha un errore.
  - Se T1 e T2 sono tipi set compatibili ed un membro almeno dell'insieme valore di E non appartiene all'intervallo chiuso specificato dal tipo base T1 si ha un errore.