Relazione Progetto Compilatori 2
SimpLanPlus
Parte II

Stefano Notari
Francesco Santilli

10 Giugno 2022

Indice

1 Esercizio 3

Per poter verificare gli errori semantici (dichiarazione multipla di variabili/-funzioni e dell'analisi degli effetti) è necessario visitare l'albero sintattico in modo da poter personalizzare i valori di ritorno. Ciò non era possibile con i listener, che permettevano solamente di ricevere una notifica all'entrata e all'uscita di una regola. Quindi abbiamo utilizzato i visitor messi a disposizione da ANTLR. Abbiamo utilizzato i seguenti visitor:

- SimpLanPlusVisitorTypeCheck: permette di verificare gli errori di tipo
- SimpLanPlusVisitorEffect: implementa l'analisi degli effetti

I log sono consultabili nella cartella **src/output/error_logger_TS.log** (dove TS indica il timestamp).

1.1 Regole semantiche per type checking

L'analisi dei tipi viene effettuata solo se non sono presenti funzioni/variabili dichiarate molteplici volte o l'utilizzo di funzioni/variabili non dichiarate implementato nell'esercizio 2. Le categoria declaration modifica l'environment, statement accede all'environment per verificare i tipi, i quali sono ritornati dalle espressioni (exp).

1.1.1 block

$$[block] = \frac{\Gamma * [] \vdash declaration : \Gamma' \quad \Gamma' \vdash statement}{\Gamma \vdash \quad \{declaration \quad statement\}}$$

Quando si entra in un blocco si aggiunge un nuovo environment e si procede con la valutazione delle dichiarazioni/statement.

1.1.2 sequence declaration

$$[declaration] = \frac{\Gamma \vdash declaration1 : \Gamma' \quad \Gamma' \vdash declaration2 : \Gamma''}{\Gamma \vdash \quad declaration1; declaration2 : \Gamma''}$$

Quando ho una sequenza di dichiarazioni procedo a valutarle in fila, passando alle valutazioni successive la lista degli environment aggiornata.

1.1.3 sequence statement

$$[statement] = \frac{\Gamma \vdash statement1 \quad \Gamma \vdash statement2}{\Gamma \vdash \quad statement1; statement2}$$

1.1.4 assignment

$$[assignment] = \frac{\begin{array}{c} ID \in dom(\Gamma) \quad \Gamma \vdash \quad exp \quad : type\_exp \\ type\_ID = \Gamma[ID] \quad type\_exp == type\_ID \end{array}}{\Gamma \vdash \quad ID = \quad exp;}$$

1.1.5 print

$$[print] = \frac{\Gamma \vdash \quad exp \quad : type}{\Gamma \vdash \quad print \quad exp;}$$

1.1.6 return

$$[ret] = \frac{\Gamma \vdash \quad exp \quad : type}{\Gamma \vdash \quad return \quad exp \quad ; : type}$$

1.1.7 ite

$$[ite\_then] = \frac{\Gamma \vdash \quad exp : type \quad \Gamma \vdash statement \quad type = TypeBool}{\Gamma \vdash \quad if \quad (\quad exp \quad) \quad statement}$$

$$[ite\_then\_else] = \frac{\begin{array}{c} \Gamma \vdash exp : type \qquad \Gamma \vdash statement\_then \\ \Gamma \vdash statement\_else \qquad type = TypeBool \end{array}}{\Gamma \vdash \quad if \quad (\quad exp \quad) \quad statement\_then \quad else \quad statement\_else}$$

1.1.8 call

$$[call] = \frac{\begin{array}{c} ID \in dom(\Gamma) \\ \Gamma \vdash f : type_1 \quad \times \quad ... \quad \times \quad type_n \rightarrow type \\ (\Gamma \vdash type_i : type_{i'})^{i \in 1..n} \qquad (type_i = type_{i'})^{i \in 1..n} \end{array}}{\Gamma \vdash \quad ID(e_1, ..., e_n) : type}$$

1.1.9 decFun

$$[decFun\_void] = \frac{ID \notin dom(head(\Gamma)) \qquad \Gamma \vdash block}{\Gamma \vdash void \quad ID \quad (T_1 \quad e_1, .., T_n \quad e_n) \quad block : \\ \Gamma[ID \rightarrow < VoidType, e_1 : T_1 * ... * e_n : T_n >]}$$

$$[decFun] = \frac{ID \notin dom(head(\Gamma)) \quad \Gamma \vdash type : type\_fun \quad \Gamma \vdash block}{\Gamma \vdash type \quad ID \quad (T_1 \quad e_1, .., T_n \quad e_n) block : \\ \Gamma[ID \rightarrow < type\_fun, e_1 : T_1 * ... * e_n : T_n >]}$$

Quando si dichiara una nuova funzione si controlla che non sia definita nell'ambiente attuale (che viene ottenuto con la funzione head che ritorna l'ambiente di testa). Oltre ad inserire il tipo della funzione si effettua una visita della dichiarazione dei parametri formali, attraverso un inserimento in una lista (che mappa il nome al tipo) e che viene inserito nella symbol table quando si effettua la visita del blocco.

1.1.10 decVar e decVar_assignment

$$[decVar] = \frac{ID \notin dom(head(\Gamma))\Gamma \vdash type : type\_id}{\Gamma \vdash type \quad ID : \Gamma[ID- > type\_id]}$$

Quando si dichiara una nuova variabile si controlla che non sia definita nell'ambiente attuale (che viene ottenuto con la funzione head che ritorna l'ambiente di testa).

$$[decVar\_assignment] = \frac{\Gamma \vdash exp : type\_exp \quad ID \notin dom(head(\Gamma)) \\ \Gamma \vdash type : type\_id \qquad type\_exp == type\_id}{\Gamma \vdash \quad type \quad ID \quad = \quad exp : \Gamma[ID \rightarrow type\_id]}$$

Γ è una lista di environment, la verifica della presenza della dichiarazione della variabile ID si effettua nel blocco in cima (quello corrente).

1.1.11 baseExp

$$[baseExp] = \frac{\Gamma \vdash exp : type}{\Gamma \vdash (\quad exp \quad) : type}$$

1.1.12 negExp

$$[negExp] = \frac{\Gamma \vdash exp : type \quad type == IntType}{\Gamma \vdash \quad -exp : type}$$

1.1.13 notExp

$$[notExp] = \frac{\Gamma \vdash exp : type \quad type == BoolType}{\Gamma \vdash \quad !exp : type}$$

1.1.14 derExp

$$[derExp] = \frac{ID \in dom(\Gamma) \quad type = \Gamma[ID]}{\Gamma \vdash ID : type}$$

1.1.15 binExp

$$[binExp] = \frac{\Gamma \vdash left : type1 \qquad \Gamma \vdash right : type2 \qquad type1 == type2 == TypeInt}{\Gamma \vdash left \quad (+ \quad | \quad - \quad | \quad * \quad | \quad /) \quad right : TypeInt}$$

$$[binExp2] = \frac{\Gamma \vdash left : type1 \qquad \Gamma \vdash right : type2 \qquad type1 == type2 == TypeBool}{\Gamma \vdash left \quad (\| \quad | \quad \&\&) \quad right : TypeBool}$$

$$[binExp3] = \frac{\Gamma \vdash left : type1 \qquad \Gamma \vdash right : type2 \qquad type1 == type2 == TypeBool}{\Gamma \vdash left \quad (< \quad | \quad <= \quad | \quad > \quad | \quad =>) \quad right : TypeBool}$$

$$[binExp4] = \frac{\Gamma \vdash left : type1 \qquad \Gamma \vdash right : type2 \qquad type1 == type2 == TypeBool}{\Gamma \vdash left \quad (! = \quad | \quad ==) \quad right : TypeBool}$$

1.1.16 valExp

$$[valExp] = \frac{}{\Gamma \vdash NUMBER : TypeInt}$$

1.1.17 boolExp

$$[boolExp] = \frac{}{\Gamma \vdash BOOL : TypeBool}$$

1.2 Regole semantiche per gli effetti

Per la gestione del passaggio per riferimento (variabile) supponiamo che la variabile sia non inizializzata, questo produce come effetto lo scarto di alcuni programmi che invocano la funzione con variabili già inizializzate. Abbiamo scelto questo approccio in quanto abbiamo riscontrato dei problemi implementativi con il metodo corretto. Inizialmente, pensavamo di effettuare la visita della dichiarazione di funzione con lo stato delle variabili presenti nella chiamata a funzione in modo da inizializzare le variabili passate per riferimento con il giusto valore ($\perp$ o Init).

```
{
    int y = 5;
    void f(var int x) {
        print(x);
    }
    f(y);
}
```

Gli stati utilizzati per questa implementazione sono:

- $\perp$: variabile dichiarata ma non inizializzata
- Init: variabile inizializzata
- RW: variabile che ha avuto un accesso in lettura
- $\top$: variabile con uno stato d'errore

La funzione update è così definita:

$$Update(value, newValue) = \begin{cases} \top & \text{value} == \perp \;\&\&\; \text{newValue} == \text{RW} \\ max(value, newValue) & \text{otherwise} \end{cases} \tag{1}$$

$$max(value, newValue) = \begin{cases} \top & \text{value} == \top \;||\; \text{newValue} == \top \\ RW & \text{value} == \text{RW} \;||\; \text{newValue} == \text{RW} \\ Init & \text{value} == \text{Init} \;||\; \text{newValue} == \text{Init} \\ \perp & \text{otherwise} \end{cases} \tag{2}$$

1.2.1 block

$$[block] = \frac{\Sigma * [] \vdash declaration : \Sigma' \quad \Sigma' \vdash statement \quad (\Sigma'[x] == RW)^{x \in head(\Sigma')}}{\Sigma \vdash \quad \{declaration \quad statement\}}$$

1.2.2 sequence statement

$$[statement] = \frac{\Sigma \vdash statement1 : \Sigma' \quad \Sigma' \vdash statement2 : \Sigma''}{\Sigma \vdash \quad statement1; statement2 : \Sigma''}$$

Ogni statement può aggiornare lo stato di utilizzo di una variabile, quindi oltre ad accedere in lettura lo stato di una variabile può anche aggiornare lo stato (per esempio se accedo in lettura a una variabile non inizializzata che genera lo stato $\top$).

1.2.3 sequence declaration

$$[declaration] = \frac{\Sigma \vdash declaration1 : \Sigma' \quad \Sigma' \vdash declaration2 : \Sigma''}{\Sigma \vdash \quad declaration1; declaration_2 : \Sigma''}$$

1.2.4 decFun

$$[decFun] = \frac{\Sigma' \vdash block \qquad appendVar(\Sigma, ID, \Sigma'[e_i])^{i=1..n}}{\begin{array}{r}\Sigma \vdash TypeID(varT_1e_1, ..., varT_me_m, T_1'e_1'..., T_n'e_n')block : \\ \Sigma * [e_1 \to \bot, ..., e_m \to \bot, e_1' \to Init, ..., e_n' \to Init]\end{array}}$$

Quando si dichiara una nuova funzione si aggiorna Σ settando i parametri formali passati per valore a Init e quelli passati per variabile (riferimento) a $\bot$, all'interno del blocco mi salvo lo stato dei parametri formali e una volta uscito dal blocco li associo all'id utilizzando appendVar.

1.2.5 decVar

$$[decVar] = \frac{}{\Sigma \vdash TypeID : \Sigma[ID- > \bot]}$$

$$[decVar\_assignment] = \frac{\Sigma \vdash exp : effect(\bot < effect < \top) update(\Sigma, ID, Init)}{\Sigma \vdash TypeID = exp}$$

Quando si dichiara una variabile si setta il suo stato a $\bot$. Quando c'è l'assegnamento si controlla che l'effetto dell'exp assegnata non sia $\bot$ nè $\top$, per poi aggiornare lo stato della variabile a Init.

1.2.6 valExp

$$[valExp] = \frac{}{\Sigma \vdash NUMBER : Init}$$

$$[boolExp] = \frac{}{\Sigma \vdash BOOL : Init}$$

1.2.7 assignment

$$[assignment] = \frac{\Sigma \vdash exp : effect(\bot < effect < \top) \qquad update(\Sigma, ID, Init)}{\Sigma \vdash ID = exp;}$$

1.2.8 print

$$[print] = \frac{\Sigma \vdash exp : effect(\bot < effect < \top)}{\Sigma \vdash exp : effect}$$

1.2.9 ret

$$[ret] = \frac{\Sigma \vdash exp : effect(\bot < effect < \top)}{\Sigma \vdash return \quad exp;}$$

1.2.10 ite

$$[ite] = \frac{\begin{array}{c}\Sigma \vdash exp : status \qquad (\bot < status < \top) \\ \Sigma \vdash statement\_then : \Sigma_1 \qquad \Sigma \vdash statement\_else : \Sigma_2\end{array}}{\Sigma \vdash if(exp) \quad statement\_then \quad else \quad statement\_else : max(\Sigma_1, \Sigma_2)}$$

Per valutare gli effetti dell'if, si valuta il ramo then e il ramo else partendo dalla stessa di ambienti (Σ). Successivamente, per ogni variabile appartenente alla lista degli ambienti prende il massimo valore ottenuto dalla valutazione del ramo then e else. Se lo statement dell'else risulta nullo il massimo sarà la lista di ambienti dello statement then.

1.2.11 call

$$[call] = \frac{update(\Sigma, u_i, setNewState(\Sigma, f, i))^{i=1...m} status = (isVoid(\Sigma, f)?\bot|Init)}{\Sigma \vdash \quad f(u_1, ..., u_m, v_1, ..., v_n) : status}$$

Cerca ogni variabile passata per riferimento all'interno dell'ambiente e aggiorna il suo stato. Se la funzione è di tipo Void ritorna $\bot$, sennò ritorna Init.

1.2.12 binExp

$$[binExp] = \frac{\begin{array}{c}\Sigma \vdash left : effect\_left \qquad \Gamma \vdash right : effect\_right \\ effect = calculateEffectBinExp(effect\_left, effect\_right)\end{array}}{\Gamma \vdash left \quad * \quad right : effect}$$

la funzione calculateEffectBinExp presi in input due stati ritorna lo stato Init se e solo se entrambi gli stati sono compresi tra Init e RW, altrimenti ritorna $\top$.

1.2.13 derExp

$$[derExp] = \frac{ID \in dom(\Sigma)(\bot < Sigma[ID] < \top)}{\Sigma \vdash ID : \Sigma[ID- > RW]}$$

1.2.14 baseExp

$$[baseExp] = \frac{\Sigma \vdash exp : effect}{\Sigma \vdash (\quad exp \quad) : effect}$$

1.2.15 negExp

$$[negExp] = \frac{\Sigma \vdash exp : effect}{\Sigma \vdash -exp : effect}$$

1.2.16 notExp

$$[notExp] = \frac{\Sigma \vdash exp : effect}{\Sigma \vdash !exp : effect}$$

1.3 Codici da verificare e discutere

Premessa: In nessun codice abbiamo trovato errori di tipo. Nel codice d'esempio numero 2 abbiamo eleminato una parentesi graffa superflua (alla riga 4) poichè ci sembrava un errore di battitura.

1.3.1 Codice 1

```
1 {
2     int a; int b; int c = 1 ;
3     if (c > 1) {
4         b = c ;
5     } else {
6         a = b ;
7     }
8 }
```

```
1 Number of errors: 2
2   line 6: Attention the variable b is not initialized
3   line 6: Attention invalid assignment of a
```

Analizzando il codice ci aspettiamo che ritorni un errore, in quando nel ramo else si tenta di fare un assegnamento con una variabile non inizializzata. a diventa $\top$ perchè gli si assegna una variabile che è $\bot$. b diventa $\top$ perchè il suo stato passa da $\bot$ a RW (e questo non è possibile).

1.3.2 Codice 2

```
1 {
2     int a; int b; int c ;
3     void f(int n, var int x) {
4         x = n;
5     }
6     f(1,a) ; f(2,b) ; f(3,c) ;
7  }
```

```
1 Number of errors: 3
2   line 1: Warning the variable a is declared but never used
3   line 1: Warning the variable b is declared but never used
4   line 1: Warning the variable c is declared but never used
```

Abbiamo eliminato una parentesi superflua che non chiudeva nessun blocco in quanto la ritenevamo un errore di battitura. Analizzando il codice ci

aspettiamo che non ci siano errori di assegnamento o di dichiarazione, ma che ritorni un avvertimento per l'utilizzo di variabili inizializzate ma non utilizzate, in quanto le variabili (a, b,c) non sono mai accedute in lettura (per assegnamento o stampa). Per contro prova abbiamo testato il codice aggiungendo la variabile x a dx dell'assegnamento e non abbiamo ottenuto nessun avvertimento.

1.3.3 Codice 3

```
{
    int a; int b; int c = 1 ;
    void h(int n, var int x, var int y, var int z){
        if (n==0) return ;
        else {
            x = y ;
            h(n-1,y,z,x) ;
        }
    }
    h(5,a,b,c) ;
}
```

```
Number of errors: 2
  line 6: Attention the variable y is not initialized
  line 6: Attention invalid assignment of x
```

Analizzando il codice ci aspettiamo di ricevere 2 errori: entrambi relativi all'assegnamento x = y. Questo perchè dopo la prima chiamata di h, la variabile y è $\perp$, quindi non può diventare RW. Inoltre non è possibile assegnare ad una variabile $\perp$ (x) un'altra variabile $\perp$ (y);

2 Esercizio 4 - Generazione codice intermedio

2.1 Gestione memoria

La gestione della memoria è composta dall'utilizzo di due record di attivazione:

1. Blocchi 2.1.1
2. Funzioni 2.1.2

Il motivo è dato dal fatto che per le chiamate a funzione l'AR è costruito dalla collaborazione tra il chiamato e il chiamante, a differenza degli altri casi.

2.1.1 Blocchi

Il record di attivazione viene gestito nel seguente modo:

1. Viene aggiunto in cima allo stack il vecchio fp
2. Valuta le espressioni in ordine inverso. Prima alloca lo spazio sullo stack, una volta terminata la costruzione dell'AR si valuta, l'eventuale, espressione di inizializzazione
3. Viene aggiunto in cima allo stack l'access link

Access Link
Variables
Old FP

Tabella 1: Record di attivazione per i blocchi

2.1.2 Funzioni

Il record di attivazione viene gestito nel seguente modo:

1. Viene aggiunto in cima allo stack il vecchio fp
2. Valuta le espressioni in ordine inverso. Se l'espressione è una variabile passata per:
   - riferimento: sullo stack viene messo il relativo indirizzo di memoria (nella symbol table viene marcata la relativa variabile per gestire l'accesso in lettura e scrittura)
   - valore: sullo stack viene messo il relativo valore
3. Valuta le dichiarazioni interne al blocco e alloca lo spazio richiesto
4. Viene aggiunto in cima allo stack l'access link (calcolato dal chiamante e passato con il registro $t2)
5. Viene aggiunto in cima allo stack il return address

Return address
Access Link
Variables
Actual Parameters
Old FP

Tabella 2: Record di attivazione per le funzioni

2.2 Symbol Table

La symbol table è composta da una lista di ambienti (che rappresentano l'annidamento dei blocchi) e per ciascuno di essi contiene le seguenti informazioni:

- STentry (variabili): contine il nome della variabile, il suo tipo (bool o int), l'offset, il livello di annidamento della variabile e se è stata passata per riferimento
- STentryFunction (funzioni): estende la classe STentry ma in più contine la label (costruita dalla concatenazione del nome della funzione e dal livello di annidamento) e la lista degli argomenti

Il calcolo dell'offset dipende dal tipo della variabile, in quanto:

- i booleani occupano 8-bit
- gli interi occupano 32-bit

2.3 Registri utilizzati

- $sp: Stackpointer, punta sempre in cima allo stack dei RdA
- $fp: Framepointer, punta dopo l'access link e serve per risalire la catena statica ed accedere alle variabili in base al loro offset.
- $ra: Return address, punta all'istruzione successiva alla chiamata di funzione.
- $al: Access link, punta dopo all'access link e serve per risalire la catena statica da un blocco all'altro.

- $a0: Accumulatore, contiene un certo valore.
- $t1: Temporaneo1, registro temporaneo usato come appoggio nelle operazioni che richiedono più di un registro
- $t2: Temporaneo2, serve nella chiamata di funzione per passare al chiamato l'access link.

2.4 Istruzioni bytecode

- lw $r1 offset($r2): carica una word da 32 bit (interi) dall'indirizzo $r2+offset in $r1.
- lw_b $r1 offset($r2): carica una word da 8 bit (booleani) dall'indirizzo $r2+offset in $r1.
- addi $r1 $r2 n: il valore di $r2 + n è memorizzato in $r1.
- add $r1 $r2 $r3: il valore di $r2+$r3 è memorizzato in $r1.
- sub $r1 $r2 $r3: il valore di $r2-$r3 è memorizzato in $r1.
- div $r1 $r2 $r3: il valore di $r2/$r3 è memorizzato in $r1.
- mult $r1 $r2 $r3: il valore di $r2*$r3 è memorizzato in $r1.
- sw $r1 offset($r2): salva una word da 32 bit (interi) contenuta in $r1 nell'indirizzo $r2+offset.
- sw_b $r1 offset($r2): salva una word da 8 bit (booleani) contenuta in $r1 nell'indirizzo $r2+offset.
- li $r1 n: salva una word da 32 bit (interi) contenuta in n in $r1.
- li_b $r1 n: salva una word da b bit (booleani) contenuta in n in $r1.
- move $r1 $r2: il valore del registro $r2 è memorizzato in $r1.
- beq $r1 $r2 label: se $r1 == $r2 allora salto alla label (con interi).
- bgt $r1 $r2 label: se $r1 >$r2 allora salto alla label.
- bge $r1 $r2 label: se $r1 >= $r2 allora salto alla label.

- beq_b $r1 $r2 label: se $r1 == $r2 allora salto alla label ma con booleani.
- branch label: salto alla label.
- label: definizione di una label.
- jtl label: salta alla label e memorizza l'indirizzo di ritorno nel registro $ra.
- jr $ra: salta all'indirizzo presente nel registro $ra (return address).
- print $r1: printa il valore di un registro.
- halt: termina la computazione.

2.5 Codici da verificare

2.5.1 Codice 1

```
{
    int x = 1;
    void f(int n){
      if (n == 0) {
          print(x) ;
      }
      else {
          x = x * n ;
          f(n-1) ;
     }
}
f(10) ;}
```

Il risultato è 3628800, cioè 10!. Ci sembra corretto in quanto la funzione sembra calcolare il fattoriale di n tramite la chiamata ricorsiva.

2.5.2 Codice 2

```
{
    int u = 1 ;
    void f(var int x, int n){
      if (n == 0) {
          print(x) ;
```

```
6         }
7         else {
8             int y = x * n ;
9             f(y,n-1) ;
10       }
11  }
12  f(u,6) ;}
```

Abbiamo dei problemi a gestire i parametri passati per riferimento nella gestione degli effetti, questo blocca l'esecuzione del codice. Per poterlo eseguire abbiamo commentato temporaneamente quella parte e, nonostante la presenza di errori relativi agli effetti (e non alla parte di generazione del codice) il risultato prodotto è quello che ci aspettavamo: cioè 720.
Ci sembra corretto in quanto la funzione chiamata come f(u,n) calcola il fattoriale di n moltiplicato per u (cioè n! * u).

2.5.3 Codice 3

```
1  {
2      void f(int m, int n){
3        if (m>n) {
4            print(m+n) ;
5        }
6        else {
7            int x = 1 ;
8            f(m+1,n+1) ;
9        }
10 }
11 f(5,4) ; }
```

Anche qui abbiamo un errore relativo all'analisi degli effetti in quanto la variabile x è dichiarata ma mai utilizzata, commentando quella parte il codice viene eseguito senza problemi. Il risultato è 9, questo perchè m è maggiore di n e stampa la loro somma (5+4). Se si chiama f(4,5) invece da errore, in quanto la chiamata ricorsiva entra in loop e continua ad allocare memoria per l'intero x "all'infinito" (cioè fino a quando non finisce il buffer, che è finito).