Condizionale Multiplo per Small20

Fabiana Chericoni

13 luglio 2020

Presentazione del costrutto

$$\text{cond } exp_1 : cmd_1, ..., exp_k : cmd_k$$

dove exp_1,...,exp_k indicano k>0 espressioni booleane e cmd_1,...,cmd_k sono i corrispondenti comandi selezionabili. Il costrutto esprime un comando che esegue il comando cmd_i con il minimo i per cui le espressioni exp_1,...,exp_{i-1}, valutate in sequenza, risultino tutte false ed exp_i risulti vera.
Supponendo che gli elementi della sequenza rispettino i *tipi* indicati sopra possono verificarsi due casi :

1. Esiste almeno un'espressione vera, e avviene l'esecuzione del comando corrispondente
2. Tutte le espressioni della sequenza sono false e non viene eseguito alcun comando

Small20

Vediamo le modifiche apportate alla sintassi astratta e concreta:

- **Sintassi astratta:**

```
Cmd2 ::= [Com] Exp Cmd
Cmd2Seq ::= Cmd2 | Cmd2[x]Cmd2Seq
Cmd ::=...  | [Cond] Cmd2Seq
```

- **Sintassi concreta:**

```
Cmd2 ::= Exp : Cmd
Cmd2Seq ::= Cmd2|Cmd2,Cmd2Seq
Cmd ::= ... |cond Cmd2Seq
```

- Non sono state apportate modifiche alla **macchina astratta AM20**.

Realizzazione in Small20

Sintassi Astratta:

```
type Cmd2=
        Com of exp*cmd
        and

Cmd2Seq=
        Cmd2 list
        and

Cmd= ...
        |Cond of cmd2Seq
```

Sintassi Concreta:

```
let toStringCmd2= (function
                    |Com(exp,cmd)-> (toStringExp exp) ^ ":" ^ (toStringCmd cmd))
                    and


toStringCmd2Seq l= match l with
         []->""
         |h::t -> if (t==[]) then (toStringCmd2 h)
                  else (toStringCmd2 h)^  ", "  ^(toStringCmd2Seq t);;


toStringCmd = (function
                ...
               |Cond(cmd2)->
                  "cond" ^ " " ^ (toStringCmd2Seq cmd2);;
```

Semantica e Sistema Y per Small20

- Aggiornamento delle **regole** del sistema $\mathtt{Sem}_{\mathtt{CMD}}$

$$\frac{\langle \mathtt{e}_1,\sigma\rangle \rightarrow |[\mathtt{bool}],\mathtt{true},\sigma_\mathtt{e}| \quad \langle \mathtt{c}_1,\sigma_\mathtt{e}\rangle \rightarrow ([\mathtt{void}],\sigma_\mathtt{f})}{\langle [\mathtt{cond}]\ \mathtt{e}_1\mathtt{c}_1\ \ldots\ \mathtt{e}_\mathtt{k}\mathtt{c}_\mathtt{k}\ ,\sigma\rangle \rightarrow ([\mathtt{void}],\sigma_\mathtt{f})}$$

$$\frac{\langle \mathtt{e}_1,\sigma\rangle \rightarrow |[\mathtt{bool}],\mathtt{false},\sigma_\mathtt{e}| \quad \langle [\mathtt{cond}]\ \mathtt{e}_2\mathtt{c}_2\ \ldots\ \mathtt{e}_\mathtt{k}\mathtt{c}_\mathtt{k},\sigma_\mathtt{e}\rangle \rightarrow ([\mathtt{void}],\sigma_\mathtt{f})}{\langle [\mathtt{cond}]\ \mathtt{e}_1\mathtt{c}_1\ \ldots\ \mathtt{e}_\mathtt{k}\mathtt{c}_\mathtt{k}\ ,\sigma\rangle \rightarrow ([\mathtt{void}],\sigma_\mathtt{f})}$$

- Aggiornamento del **Sistema Y**

$$\frac{\langle \mathtt{e}_\mathtt{i},\mathtt{Y}_\rho\rangle \rightarrow_\mathtt{Y} ([\mathtt{bool}],\mathtt{Y}_\rho)\ \forall 1\leq \mathtt{i}\leq \mathtt{k} \quad \langle \mathtt{c}_\mathtt{i},\mathtt{Y}_\rho\rangle \rightarrow_\mathtt{Y} ([\mathtt{void}],\mathtt{Y}_\rho)\ \forall 1\leq \mathtt{i}\leq \mathtt{k}}{\langle [\mathtt{cond}]\ \mathtt{e}_1\mathtt{c}_1\ \ldots\ \mathtt{e}_\mathtt{k}\mathtt{c}_\mathtt{k}\ ,\mathtt{Y}_\rho\rangle \rightarrow ([\mathtt{void}],\mathtt{Y}_\rho)}$$

Gestione errori di tipo

In seguito all'aggiornamento del sistema Y i possibili errori di tipo sono due:

1. **E31**: dovuto alla presenza di almeno un espressione non booleana

$$\frac{\langle \mathtt{e_i}, \mathtt{Y}_\rho \rangle \rightarrow_{\mathtt{Y}} (\mathtt{t_i}, \mathtt{Y}_\rho) \ \forall 1 \leq i \leq k \qquad \exists\ \mathtt{i} : \mathtt{t_i} \neq [\mathtt{bool}]}{\langle [\mathtt{cond}]\ \mathtt{e_1 c_1}\ \ldots\ \mathtt{e_k c_k}\ , \mathtt{Y}_\rho \rangle \rightarrow ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

2. **E32**: dato dalla presenza di almeno un comando il cui tipo è diverso da Void

$$\frac{\langle \mathtt{c_i}, \mathtt{Y}_\rho \rangle \rightarrow_{\mathtt{Y}} (\mathtt{t_i}, \mathtt{Y}_\rho) \ \forall 1 \leq i \leq k \qquad \exists\ \mathtt{i} : \mathtt{t_i} \neq [\mathtt{Void}]}{\langle [\mathtt{Cond}]\ \mathtt{e_1 c_1}\ \ldots\ \mathtt{e_k c_k}\ , \mathtt{Y}_\rho \rangle \rightarrow ([\mathtt{Terr}], \mathtt{Y}_\rho)}$$

Implementazione in ocaml

Inserimento del costrutto [Cond] in $\mathtt{Sem}_{\mathtt{CMD}}$:

```
|Cond(cmd2) -> (
        match cmd2 with
         |Com(exp,cmd)::tail ->
           (match expSem exp (rho,mu) with
                |(Bool, Bval True,sg1) ->
                 (match conexp tail sg1 with
                    |(True,sg2)->
                      (match cmdSem cmd sg2 with
                            |(Void,sg2) -> (Void,sg2)
                            |_ ->let msg = "E32: " ^(toStringCmd 0 cmd)
                                           in raise(TypeErrorC msg)))

                |(Bool,Bval False,sg1) ->(let c1=Cond(tail) in cmdSem c1 sg1)

                |_->(let msg= "E31: " ^(toStringExp exp) in raise(TypeErrorC msg)))

        |[]->(Void,(rho,mu)))
```

Funzione di controllo delle espressioni

```
conexp cmd2 (rho,(Store(d,g)as mu)) =
     (match cmd2 with
                |Com(exp,cmd)::tail ->
                  (match expSem exp (rho,mu) with
                        |(Bool,Bval True,sg1) -> conexp tail sg1
                        |(Bool,Bval False,sg1) -> conexp tail sg1
                        |_ -> (let msg= "E31: " ^ (toStringExp exp)
                                in raise(TypeErrorC msg) ))
                |[] -> (True,(rho,mu)))
        and
```

Come si nota dall'implementazione della semantica, è stato necessario definire una nuova funzione chiamata **conexp** (controllo espressioni).
Quando troviamo il minimo i per cui tutte le espressioni $exp_1 ... exp_{i-1}$ risultano false e exp_i vera, **conexp** controlla che non vi siano errori di tipo nelle espressioni successive prima di poter effettuare il comando corrispondente cmd_i.
Quest'ultime, in accordo al sistema dei tipi presentato, devono essere tutte booleane. Dopo l'applicazione di conexp alla restante parte della lista $exp_{i+1} : cmd_{i+1}, ..., exp_k : cmd_k$ è possibile effettura il comando cmd_i solo in caso di assenza di errori.

Funzione di controllo delle espressioni

Notiamo come, in assenza della nostra funzione di controllo, venga eseguito il comando nonostante vi sia un errore di tipo dovuto alla presenza di un'espressione non booleana:

Presenza della funzione **conexp**

```
Program es1{
   var int x = -19;
   var int fx;
cond (x + 0): fx = x, (x == 0): fx = 0, (x < 0): fx = -x;
}

- : unit = ()
# run p;;
Exception: TypeErrorC "E31: (x + 0)".
```

Assenza della funzione **conexp**

```
Program es1{
   var int x = 0;
   var int fx;
cond (x > 0): fx = x, (x == 0): fx = 0, (x + 0): fx = x;
}

- : unit = ()

=============== Traccia del Programma es1 ===========
=====================================================
Stack: [fx/(int,L1); x/(int,L0)]
Store: [L0<-0,L1<-0]
=====================================================
==================== Traccia: Fine ==================
```

Verifica del codice ed esempi

```
let stm=StmD(Var(Int,"x",Neg(N 19)));;
let stm1=StmD(VarN(Int,"fx"));;
let stm2=StmC(Cond([Com(GT(Val "x",N 0),Cmd(Upd(Val "fx", Val "x")));
---------Com(Eq(Val "x",N 0),Cmd(Upd(Val "fx",N 0)));
---------Com(LT(Val "x",N 0),Cmd(Upd(Val "fx",Val "x")))]));;
let p=Prog("es1",Seq[stm;stm1;stm2]);;
printProg p;;
run p;;
```

```
Program es1{
   var int x = -19;
   var int fx;
cond (x > 0): fx = x, (x == 0): fx = 0, (x < 0): fx = -x;
}

- : unit = ()
# run p;;

=============== Traccia del Programma es1 ===========
====================================================
Stack: [fx/(int,L1); x/(int,L0)]
Store: [L0<--19,L1<-19]
====================================================
====================== Traccia: Fine ===============
- : unit = ()
```

Figura: Esempio in cui esite almeno un' espressione vera

Verifica del codice ed esempi

```
let stm=StmD(Var(Int,"x",N 0));;
let stm1=StmD(VarN(Int,"fx"));;
let stm2=StmC(Cond([Com(GT(Val "x",N 0),Cmd(Upd(Val "fx", Val "x")));
---------Com(LT(Val "x",N 0),Cmd(Upd(Val "fx",Val "x")))]));;
let p=Prog("es1",Seq[stm;stm1;stm2]);;
printProg p;;
run p;;
```

```
Program es1{
   var int x = 0;
   var int fx;
cond (x > 0): fx = x, (x < 0): fx = -x;
}

- : unit = ()
# run p;;

============== Traccia del Programma es1 ===========
================================================
Stack: [fx/(int,L1); x/(int,L0)]
Store: [L0<-0,L1<-Undef]
================================================
==================== Traccia: Fine ================
- : unit = ()
```

Figura: Esempio in cui le espressioni valutate sono tutte false

Segnalazione degli errori

Errore di tipo

```
let stm=StmD(Var(Int,"x",N 0));;
let stm1=StmD(VarN(Int,"fx"));;
let stm2=StmC(Cond([Com(GT(Val "x",N 0),Cmd(Upd(Val "fx", Val "x")));
---------Com(Eq(Val "x",N 0),Cmd(Upd(Val "fx",N 0)));
---------Com(Plus(Val "x",N 0),Cmd(Upd(Val "fx",Val "x")))]));;
let p=Prog("es1",Seq[stm;stm1;stm2]);;
printProg p;;
run p;;
```

```
Program es1{
   var int x = 0;
   var int fx;
cond (x > 0): fx = x, (x == 0): fx = 0, (x + 0): fx = -x;
}

- : unit = ()
Exception: TypeErrorC "E31: (x + 0)".
```

Considerazioni finali sul costrutto

Il costrutto di Condizionale Multiplo, in alcuni casi può essere considerato come una serie finita di comandi if-Else e if-Then annidati.
Infatti scrivere cond $exp_1 : cmd_1, ..., exp_k : cmd_k$ equivale a eseguire k-1 comandi IfE e a seguire un ultimo ifT contenente l'ultima coppia della lista (exp_k, cmd_k).
Una differenza tra i due comandi è presente:
Abbiamo osservato che cond non effettua alcun comando presente nella lista se esiste almeno un'espressione non booleane, ovunque essa si trovi. Invece, la successione annidata di if esegue il comando cmd_i con il minimo i per cui le espressioni exp_1,...,exp_{i-1}, valutate in sequenza, risultino tutte false ed exp_i risulti vera, come cond, però non si occupa di controllare che le espressioni successive a exp_i siano effettivamente booleane.
Dunque in alcuni casi i due comandi non sono equivalenti, l'esecuzione di cond potrebbe generare un errore mentre l'esecuzione della sequenza degli if no.

Relazioni con altri costrutti di Small20

Riprendiamo il primo esempio di utilizzo del comando cond, riportato sopra, e osserviamo che è possibile ottenere lo stesso risultato tramite la concatenazione di comandi if :

Risultato tramite cond

```
Program es1{
   var int x = -19;
   var int fx;
cond (x > 0): fx = x, (x == 0): fx = 0, (x < 0): fx = -x;
}

- : unit = ()
# run p;;

=============== Traccia del Programma es1 ===========
====================================================
Stack: [fx/(int,L1); x/(int,L0)]
Store: [L0<--19,L1<-19]
====================================================
==================== Traccia: Fine ==================
- : unit = ()
```

Risultato tramite if annidati

```
Program es{
   var int x = -19;
   var int fx;
   if (x > 0) fx = x;
   else if (x == 0) fx = x;
else if (x < 0) fx = -x;
}

- : unit = ()
# run p;;

=============== Traccia del Programma es ===========
====================================================
Stack: [fx/(int,L1); x/(int,L0)]
Store: [L0<--19,L1<-19]
====================================================
==================== Traccia: Fine ==================
- : unit = ()
```

Relazioni con altri costrutti di Small20

Riprendiamo l'esempio precedente (slide 10), relativo agli errori dei tipi e osserviamo come il comando dato dalla sequenza degli if non riconosca l'errore di tipo E31:

Risultato tramite cond

```
Program es1{
   var int x = -19;
   var int fx;
cond (x + 0): fx = x, (x == 0): fx = 0, (x < 0): fx = -x;
}

- : unit = ()
# run p;;
Exception: TypeErrorC "E31: (x + 0)".
```

Risultato tramite if annidati

```
Program es1{
   var int x = 0;
   var int fx;
   if (x > 0) fx = x;
   else if (x == 0) fx = 0;
else if (x + 0) fx = x;
}

- : unit = ()

=============== Traccia del Programma es1 ===========
================================================
Stack: [fx/(int,L1); x/(int,L0)]
Store: [L0<-0,L1<-0]
================================================
==================== Traccia: Fine ==================
- : unit = ()
```