Autoincremento pre/post-fisso per Small20

Francesca Rizzo

29 maggio 2020

Autoincremento pre/post-fisso per Small20

`++dex, dex++`

Autoincremento pre/post-fisso per Small20

++dex, dex++

Data dex espressione denotabile di Small20 introduciamo i due costrutti:

- ++dex: operatore di pre-incremento - incrementa di 1 il valore associato a dex e restituisce il valore di dex aggiornato.

```
int i = 1;
int a = ++i;
```

Restituisce i = 2, a = 2

- dex++: operatore di post-incremento - incrementa di 1 il valore associato a dex e restituisce il valore di dex prima dell'incremento.

```
int i = 1;
int a = i++;
```

Restituisce i = 2, a = 1

Autoincremento pre/post-fisso per Small20

++dex, dex++

Data dex espressione denotabile di Small20 introduciamo i due costrutti:

- ++dex: operatore di pre-incremento - incrementa di 1 il valore associato a dex e restituisce il valore di dex aggiornato.

```
int i = 1;
int a = ++i;
```

  Restituisce i = 2, a = 2

- dex++: operatore di post-incremento - incrementa di 1 il valore associato a dex e restituisce il valore di dex prima dell'incremento.

```
int i = 1;
int a = i++;
```

  Restituisce i = 2, a = 1

Autoincremento pre/post-fisso per Small20

++dex, dex++

Data dex espressione denotabile di Small20 introduciamo i due costrutti:

- ++dex: operatore di pre-incremento - incrementa di 1 il valore associato a dex e restituisce il valore di dex aggiornato.

```
int i = 1;
int a = ++i;
```

  Restituisce i = 2, a = 2

- dex++: operatore di post-incremento - incrementa di 1 il valore associato a dex e restituisce il valore di dex prima dell'incremento.

```
int i = 1;
int a = i++;
```

  Restituisce i = 2, a = 1

Small20

- Sintassi Astratta: AST in Ocaml
  `Exp ::== ... | [++]Exp | Exp[++]`
- Sintassi Concreta: Una CFG per Small20
  `ExpA ::== ... | ++ DExp | DExp ++`
  `DExp ::== ide | ide [ExpA2] | ϵ`
- Sistema dei tipi Y:
- Abstract Machine: AM20 - Stato (Invariato)

Small20

- Sintassi Astratta: AST in Ocaml
  `Exp ::== ... | [++]Exp | Exp[++]`
- Sintassi Concreta: Una CFG per Small20
  `ExpA ::== ... | ++ DExp | DExp ++`
  `DExp ::== ide | ide [ExpA2]| ` ϵ
- Sistema dei tipi Y:
- Abstract Machine: AM20 - Stato (Invariato)

Small20

- Sintassi Astratta: AST in Ocaml
  `Exp ::== ... | [++]Exp | Exp[++]`
- Sintassi Concreta: Una CFG per Small20
  `ExpA ::== ... | ++ DExp | DExp ++`
  `DExp ::== ide | ide [ExpA2]| ` ϵ
- Sistema dei tipi Y:
- Abstract Machine: AM20 - Stato (Invariato)

Small20

- Sintassi Astratta: AST in Ocaml
  `Exp ::== ... | [++]Exp | Exp[++]`
- Sintassi Concreta: Una CFG per Small20
  `ExpA ::== ... | ++ DExp | DExp ++`
  `DExp ::== ide | ide [ExpA2] | ` ϵ
- Sistema dei tipi Y:

$$\text{Ypre:}\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{DY}} (\texttt{[mut] t}, \texttt{Y}_\rho) \quad \texttt{t} = \texttt{[int]}}{\langle \texttt{[++]e}, \texttt{Y}_\rho \rangle \rightarrow (\texttt{t}, \texttt{Y}_\rho)} \qquad \text{Ypost:}\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{DY}} (\texttt{[mut] t}, \texttt{Y}_\rho) \quad \texttt{t} = \texttt{[int]}}{\langle \texttt{e[++]}, \texttt{Y}_\rho \rangle \rightarrow (\texttt{t}, \texttt{Y}_\rho)}$$

- Abstract Machine: AM20 - Stato (Invariato)

Small20

- Sintassi Astratta: AST in Ocaml
  `Exp ::== ... | [++]Exp | Exp[++]`
- Sintassi Concreta: Una CFG per Small20
  `ExpA ::== ... | ++ DExp | DExp ++`
  `DExp ::== ide | ide [ExpA2] | ` ϵ
- Sistema dei tipi Y:

$$\text{Ypre:}\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{DY}} (\texttt{[mut] t}, \texttt{Y}_\rho) \quad \texttt{t} = \texttt{[int]}}{\langle \texttt{[++]e}, \texttt{Y}_\rho \rangle \rightarrow (\texttt{t}, \texttt{Y}_\rho)} \qquad \text{Ypost:}\ \frac{\langle \texttt{e}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{DY}} (\texttt{[mut] t}, \texttt{Y}_\rho) \quad \texttt{t} = \texttt{[int]}}{\langle \texttt{e[++]}, \texttt{Y}_\rho \rangle \rightarrow (\texttt{t}, \texttt{Y}_\rho)}$$

- Abstract Machine: AM20 - Stato (Invariato)

Small20

- Sintassi Astratta: AST in Ocaml
  `Exp ::== ... | [++]Exp | Exp[++]`
- Sintassi Concreta: Una CFG per Small20
  `ExpA ::== ... | ++ DExp | DExp ++`
  `DExp ::== ide | ide [ExpA2] | ` ϵ
- Sistema dei tipi Y:
- Abstract Machine: AM20 - Stato (Invariato)

```
exp =
        ...
        | PreIncr of exp
        | PostIncr of exp
        ...
        and

toStringExp = (function
        ...
        | PreIncr exp -> "(++"^(toStringExp exp)^")"
        | PostIncr exp -> "("^(toStringExp exp)^"++)"
        ...
        and
```

Strutture esprimibili

- `i++` o `++i`: incrementa di uno il valore della variabile `i` (usato ad esempio come statement iterativo nei cicli `for`, più espressivo)
- Può essere usato anche all'interno di espressioni aritmetiche: ad esempio `a + (++i)` o `a + (i++)`.
- In generale però `++dex` non è equivalente a `dex = dex + 1` (mancanza di Trasparenza Referenziale), ad esempio

```
int b = 0;
a[(b=b+1)] = a[(b=b+1)] +1;    /* a[2] = a[1]+1 */

int b = 0;
++a[(++b)];                    /* a[1] = a[1]+1 */
```

Strutture esprimibili

- `i++` o `++i`: incrementa di uno il valore della variabile `i` (usato ad esempio come statement iterativo nei cicli `for`, più espressivo)
- Può essere usato anche all'interno di espressioni aritmetiche: ad esempio `a + (++i)` o `a + (i++)`.
- In generale però ++dex non è equivalente a dex = dex + 1 (mancanza di Trasparenza Referenziale), ad esempio

```
int b = 0;
a[(b=b+1)] = a[(b=b+1)] +1;     /* a[2] = a[1]+1 */

int b = 0;
++a[(++b)];                     /* a[1] = a[1]+1 */
```

Strutture esprimibili

- `i++` o `++i`: incrementa di uno il valore della variabile `i` (usato ad esempio come statement iterativo nei cicli `for`, più espressivo)
- Può essere usato anche all'interno di espressioni aritmetiche: ad esempio `a + (++i)` o `a + (i++)`.
- In generale però `++dex` non è equivalente a `dex = dex + 1` (mancanza di Trasparenza Referenziale), ad esempio

```
int b = 0;
a[(b=b+1)] = a[(b=b+1)] +1;     /* a[2] = a[1]+1 */
```

```
int b = 0;
++a[(++b)];                     /* a[1] = a[1]+1 */
```

Regole d'inferenza

Regole $\mathtt{Sem_{EXP}}$ per $\mathtt{Exp::==[++]Exp \mid Exp[++]}$

$$\text{Xpre:}\ \frac{\begin{array}{c}\langle \mathtt{e},\sigma\rangle \to_{\mathtt{Den}} \lfloor \mathtt{[mut]\ t}, \mathtt{loc_t}, \sigma_1 \rfloor \\ \mathtt{t} = \mathtt{[int]}\quad \sigma_1 = (\rho_1,\mu_1)\quad \mu_1(\mathtt{loc_t}) = \mathtt{v_t} \\ \mathtt{v_F} = \mathtt{v_t[+]1}\quad \mu_1[\mathtt{loc_t} \leftarrow \mathtt{v_F}] = \mu_F\end{array}}{\langle \mathtt{[++]e},\sigma\rangle \to \lfloor \mathtt{t}, \mathtt{v_F}, (\rho_1,\mu_F)\rfloor} \qquad \text{Xpost:}\ \frac{\begin{array}{c}\langle \mathtt{e},\sigma\rangle \to_{\mathtt{Den}} \lfloor \mathtt{[mut]\ t}, \mathtt{loc_t}, \sigma_1 \rfloor \\ \mathtt{t} = \mathtt{[int]}\quad \sigma_1 = (\rho_1,\mu_1)\quad \mu_1(\mathtt{loc_t}) = \mathtt{v_t} \\ \mathtt{v_F} = \mathtt{v_t[+]1}\quad \mu_1[\mathtt{loc_t} \leftarrow \mathtt{v_F}] = \mu_F\end{array}}{\langle \mathtt{e[++]},\sigma\rangle \to \lfloor \mathtt{t}, \mathtt{v_t}, (\rho_1,\mu_F)\rfloor}$$

Sistema Y:

$$\text{Ypre:}\ \frac{\begin{array}{c}\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \\ \mathtt{t} = \mathtt{[int]}\end{array}}{\langle \mathtt{[++]e}, \mathtt{Y}_\rho\rangle \to (\mathtt{t}, \mathtt{Y}_\rho)} \qquad \text{Ypost:}\ \frac{\begin{array}{c}\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \\ \mathtt{t} = \mathtt{[int]}\end{array}}{\langle \mathtt{e[++]}, \mathtt{Y}_\rho\rangle \to (\mathtt{t}, \mathtt{Y}_\rho)}$$

Gestione degli errori di tipo:

$$\text{E37:}\ \frac{\begin{array}{c}\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \\ \mathtt{t} \neq \mathtt{[int]}\end{array}}{\langle \mathtt{[++]e}, \mathtt{Y}_\rho\rangle \to (\mathtt{[terr]}, \mathtt{Y}_\rho)} \qquad \text{E38:}\ \frac{\begin{array}{c}\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \\ \mathtt{t} \neq \mathtt{[int]}\end{array}}{\langle \mathtt{e[++]}, \mathtt{Y}_\rho\rangle \to (\mathtt{[terr]}, \mathtt{Y}_\rho)}$$

Regole d'inferenza

Regole $\mathtt{Sem_{EXP}}$ per $\mathtt{Exp::==[++]Exp \mid Exp[++]}$

$$\text{Xpre:}\ \frac{\langle \mathtt{e}, \sigma\rangle \to_{\mathtt{Den}} \lfloor \mathtt{[mut]\ t}, \mathtt{loc_t}, \sigma_1 \rfloor \quad \mathtt{t} = \mathtt{[int]} \quad \sigma_1 = (\rho_1, \mu_1) \quad \mu_1(\mathtt{loc_t}) = \mathtt{v_t} \quad \mathtt{v_F} = \mathtt{v_t[+]1} \quad \mu_1[\mathtt{loc_t} \leftarrow \mathtt{v_F}] = \mu_F}{\langle \mathtt{[++]e}, \sigma\rangle \to \lfloor \mathtt{t}, \mathtt{v_F}, (\rho_1, \mu_F)\rfloor}$$

$$\text{Xpost:}\ \frac{\langle \mathtt{e}, \sigma\rangle \to_{\mathtt{Den}} \lfloor \mathtt{[mut]\ t}, \mathtt{loc_t}, \sigma_1 \rfloor \quad \mathtt{t} = \mathtt{[int]} \quad \sigma_1 = (\rho_1, \mu_1) \quad \mu_1(\mathtt{loc_t}) = \mathtt{v_t} \quad \mathtt{v_F} = \mathtt{v_t[+]1} \quad \mu_1[\mathtt{loc_t} \leftarrow \mathtt{v_F}] = \mu_F}{\langle \mathtt{e[++]}, \sigma\rangle \to \lfloor \mathtt{t}, \mathtt{v_t}, (\rho_1, \mu_F)\rfloor}$$

Sistema Y:

$$\text{Ypre:}\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \quad \mathtt{t} = \mathtt{[int]}}{\langle \mathtt{[++]e}, \mathtt{Y}_\rho\rangle \to (\mathtt{t}, \mathtt{Y}_\rho)}$$

$$\text{Ypost:}\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \quad \mathtt{t} = \mathtt{[int]}}{\langle \mathtt{e[++]}, \mathtt{Y}_\rho\rangle \to (\mathtt{t}, \mathtt{Y}_\rho)}$$

Gestione degli errori di tipo:

$$\text{E37:}\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \quad \mathtt{t} \neq \mathtt{[int]}}{\langle \mathtt{[++]e}, \mathtt{Y}_\rho\rangle \to (\mathtt{[terr]}, \mathtt{Y}_\rho)}$$

$$\text{E38:}\ \frac{\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \quad \mathtt{t} \neq \mathtt{[int]}}{\langle \mathtt{e[++]}, \mathtt{Y}_\rho\rangle \to (\mathtt{[terr]}, \mathtt{Y}_\rho)}$$

Regole d'inferenza

Regole $\mathtt{Sem_{EXP}}$ per $\mathtt{Exp::==[++]Exp \mid Exp[++]}$

Xpre: $$\frac{\langle \mathtt{e}, \sigma\rangle \to_{\mathtt{Den}} \lfloor \mathtt{[mut]\ t}, \mathtt{loc_t}, \sigma_1 \rfloor \quad \mathtt{t} = \mathtt{[int]} \quad \sigma_1 = (\rho_1, \mu_1) \quad \mu_1(\mathtt{loc_t}) = \mathtt{v_t} \quad \mathtt{v_F} = \mathtt{v_t[+]1} \quad \mu_1[\mathtt{loc_t} \leftarrow \mathtt{v_F}] = \mu_F}{\langle \mathtt{[++]e}, \sigma\rangle \to \lfloor \mathtt{t}, \mathtt{v_F}, (\rho_1, \mu_F)\rfloor}$$

Xpost: $$\frac{\langle \mathtt{e}, \sigma\rangle \to_{\mathtt{Den}} \lfloor \mathtt{[mut]\ t}, \mathtt{loc_t}, \sigma_1 \rfloor \quad \mathtt{t} = \mathtt{[int]} \quad \sigma_1 = (\rho_1, \mu_1) \quad \mu_1(\mathtt{loc_t}) = \mathtt{v_t} \quad \mathtt{v_F} = \mathtt{v_t[+]1} \quad \mu_1[\mathtt{loc_t} \leftarrow \mathtt{v_F}] = \mu_F}{\langle \mathtt{e[++]}, \sigma\rangle \to \lfloor \mathtt{t}, \mathtt{v_t}, (\rho_1, \mu_F)\rfloor}$$

Sistema Y:

Ypre: $$\frac{\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \quad \mathtt{t} = \mathtt{[int]}}{\langle \mathtt{[++]e}, \mathtt{Y}_\rho\rangle \to (\mathtt{t}, \mathtt{Y}_\rho)}$$

Ypost: $$\frac{\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \quad \mathtt{t} = \mathtt{[int]}}{\langle \mathtt{e[++]}, \mathtt{Y}_\rho\rangle \to (\mathtt{t}, \mathtt{Y}_\rho)}$$

Gestione degli errori di tipo:

E37: $$\frac{\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \quad \mathtt{t} \neq \mathtt{[int]}}{\langle \mathtt{[++]e}, \mathtt{Y}_\rho\rangle \to (\mathtt{[terr]}, \mathtt{Y}_\rho)}$$

E38: $$\frac{\langle \mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{[mut]\ t}, \mathtt{Y}_\rho) \quad \mathtt{t} \neq \mathtt{[int]}}{\langle \mathtt{e[++]}, \mathtt{Y}_\rho\rangle \to (\mathtt{[terr]}, \mathtt{Y}_\rho)}$$

Implementazione: interprete Small20

```
expSem exp (rho,(Store(d,g)as mu)) =
   match exp with
      ...
      |PreIncr exp ->
         (match (dexpSem exp (rho, mu)) with
            |(Mut Int, loct, (rho1, mu1))
               ->(try (let v = mTOe(getStore mu1 loct) in
                       let vF = dispatcher v (Ival 1) ("plus",2,[Int;Int]) in
                       let muF = upd mu1 loct (eTOm vF) in
                       (Int, vF, (rho1,muF))) with
                                        |(IllegalAddress _)
                                           -> (let msg = "expSem: wrong Mutable/location found" in
                                                 raise(SystemErrorE(msg^(toStringExp exp)))))
            |(Mut t, loct, (rho1, mu1))
                  -> raise(TypeErrorE("E37: expSem - " ^ (toStringExp exp))) (*t not Int in pre*)
            |_ -> let msg = "Unexpected assign form for " in
                     raise(SystemErrorE(msg ^ (toStringExp exp)))   )
      |PostIncr exp ->
         (match (dexpSem exp (rho, mu)) with
            |(Mut Int, loct, (rho1, mu1))
               ->(try (let v = mTOe(getStore mu1 loct) in
                       let vF = dispatcher v (Ival 1) ("plus",2,[Int;Int]) in
                       let muF = upd mu1 loct (eTOm vF) in
                       (Int, v, (rho1,muF))) with
                                        |(IllegalAddress _)
                                           -> (let msg = "expSem: wrong Mutable/location found" in
                                                 raise(SystemErrorE(msg^(toStringExp exp)))))
            |(Mut t, loct, (rho1, mu1))
                  -> raise(TypeErrorE("E38: expSem - " ^ (toStringExp exp))) (*t not Int in post*)
            |_ -> let msg = "Unexpected assign form for " in
                     raise(SystemErrorE(msg ^ (toStringExp exp)))   )
```

Verifica del codice ed esempi

Pre-Incremento

```
let sd1 = StmD(Var(Int, "x", N 0));;
let sd2 = StmD(Var(Int, "a", N 1));;
let sd3 = StmD(Var(Int, "b", N 2));;
let preI = PreIncr(Val "x");;
let sc1 = StmC(Cmd(preI));;
let sc2 = StmC(Cmd(Upd(Val "a", preI)));;
let sc3 = StmC(Cmd(Upd(Val "b", Plus(Val "a",
↪  preI))));;
let sseq = Seq[sd1;sd2;sd3;sc1;sc2;sc3];;
let p = Prog("PreIncr", sseq);;
printProg p;;
run p;;
```

```
Program PreIncr{
   var int x = 0;
   var int a = 1;
   var int b = 2;
   ++x;
   a = (++x);
   b = (a + (++x));
}
- : unit = ()
======== Traccia del Programma PreIncr ========
================================================
Stack: [b/(int,L2); a/(int,L1); x/(int,L0)]
Store: [L0<-3,L1<-2,L2<-5]
================================================
================ Traccia: Fine ================
- : unit = ()
```

Post-Incremento

```
let sd1 = StmD(Var(Int, "x", N 0));;
let sd2 = StmD(Var(Int, "a", N 1));;
let sd3 = StmD(Var(Int, "b", N 2));;
let postI = PostIncr(Val "x");;
let sc1 = StmC(Cmd(postI));;
let sc2 = StmC(Cmd(Upd(Val "a", postI)));;
let sc3 = StmC(Cmd(Upd(Val "b", Plus(Val "a",
↪  postI))));;
let sseq = Seq[sd1;sd2;sd3;sc1;sc2;sc3];;
let p = Prog("PostIncr", sseq);;
printProg p;;
run p;;
```

```
Program PostIncr{
   var int x = 0;
   var int a = 1;
   var int b = 2;
   x++;
   a = (x++);
   b = (a + (x++));
}
- : unit = ()
======= Traccia del Programma PostIncr ========
================================================
Stack: [b/(int,L2); a/(int,L1); x/(int,L0)]
Store: [L0<-3,L1<-1,L2<-3]
================================================
================ Traccia: Fine ================
- : unit = ()
```

Verifica del codice ed esempi

Errore di tipo:

Pre-Incremento

```
let d1 = Var(Bool, "x", B True);;
let e1 = PreIncr(Val "x");;
let sseq = Seq[StmD(d1); StmC(Cmd(e1))];;
let p = Prog("PreError", sseq);;
printProg p;;
run p;;
```

```
Program PreError{
   var bool x = true;
   ++x;
}
- : unit = ()
Exception: TypeErrorE "E37: expSem - x".
```

Post-Incremento

```
let d1 = Var(Bool, "x", B True);;
let e1 = PostIncr(Val "x");;
let sseq = Seq[StmD(d1); StmC(Cmd(e1))];;
let p = Prog("PostError",sseq );;
printProg p;;
run p;;
```

```
Program PostError{
   var bool x = true;
   x++;
}
- : unit = ()
Exception: TypeErrorE "E38: expSem - x".
```

```
let sd1 = StmD(Var(Int, "s", N 0));;
let sd2 = StmD(Var(Int, "sD", N 0));;
let stm1 = StmD(Var(Int, "i", N 0));;
let expG = LT(Val"i", N 9);;
let er = Plus(Plus(Val "s",PostIncr(Val "i")), Val "i");;
let stm2 = StmC(Cmd(PreIncr(Val "i")));;
let c0 = Cmd(Upd(Val "s", er));;
let c1 = Cmd(Upd(Val "sD", Plus(Val "sD",Val "i")));;
let cmd = For(stm1, expG, stm2, SeqC(c0,c1));;
let sseq = Seq[sd1; sd2; StmC(cmd)];;
let p = Prog("Sum-SumOdd", sseq);;
printProg p;;
run p;;
```

```
Program Sum-SumOdd{
   var int s = 0;
   var int sD = 0;
   for (var int i = 0;(i < 9);++i)
              {
               s = ((s + (i++)) + i);
               sD = (sD + i);
              };
}
- : unit = ()
======= Traccia del Programma Sum-SumOdd ======
================================================
Stack: [i/(int,L2); sD/(int,L1); s/(int,L0)]
Store: [L0<-45,L1<-25,L2<-10]
================================================
================ Traccia: Fine ================
- : unit = ()
```

Il seguente programma calcola, dato N dispari (nel nostro caso $N = 9$):

- la somma s dei numeri minori uguali a N;
- la somma sD dei dispari minori uguali a N.

Detto i il valore di i e s quello di s, osserviamo che il comando s=(s+(i++))+(i) calcola effettivamente $s + i + (i + 1)$. Infatti $e_1 + e_2$ valuta prima e_1=s+(i++) e poi e_2=i, quindi

- valutando e_1 restituisce $s + i$ ed incrementa il valore di i di 1;
- e_2=i adesso è $i + 1$.

Poteva essere sostituito con (s+i)+(++i).

```
let sd1 = StmD(Var(Int, "s", N 0));;
let sd2 = StmD(Var(Int, "sD", N 0));;
let stm1 = StmD(Var(Int, "i", N 0));;
let expG = LT(Val"i", N 9);;
let er = Plus(Plus(Val "s",PostIncr(Val "i")), Val "i");;
let stm2 = StmC(Cmd(PreIncr(Val "i")));;
let c0 = Cmd(Upd(Val "s", er));;
let c1 = Cmd(Upd(Val "sD", Plus(Val "sD",Val "i")));;
let cmd = For(stm1, expG, stm2, SeqC(c0,c1));;
let sseq = Seq[sd1; sd2; StmC(cmd)];;
let p = Prog("Sum-SumOdd", sseq);;
printProg p;;
run p;;
```

Il seguente programma calcola, dato N dispari (nel nostro caso $N = 9$):

- la somma s dei numeri minori uguali a N;
- la somma sD dei dispari minori uguali a N.

```
Program Sum-SumOdd{
   var int s = 0;
   var int sD = 0;
   for (var int i = 0;(i < 9);++i)
              {
               s = ((s + (i++)) + i);
               sD = (sD + i);
              };
}
- : unit = ()
======= Traccia del Programma Sum-SumOdd ======
================================================
Stack: [i/(int,L2); sD/(int,L1); s/(int,L0)]
Store: [L0<-45,L1<-25,L2<-10]
================================================
================ Traccia: Fine ================
- : unit = ()
```

Detto i il valore di `i` e s quello di `s`, osserviamo che il comando `s=(s+(i++))+(i)` calcola effettivamente $s + i + (i + 1)$.
Infatti $e_1 + e_2$ valuta prima e_1=`s+(i++)` e poi e_2=`i`, quindi

- valutando e_1 restituisce $s + i$ ed incrementa il valore di i di 1:
- e_2=`i` adesso è $i + 1$.

Poteva essere sostituito con `(s+i)+(++i)`.

```
let d1 = Array(Int, "A", 5);;
let sD = StmD(Var(Int, "i", N 0));;
let eG = LT(Val "i", N 5);;
let sI = StmC(Cmd(PreIncr(Val "i")));;
let c1 = Cmd(Upd(GetArrow("A", Val "i"), Val "i"));;
let cmd1 = For(sD,eG,sI,c1);;
let p = Prog("Vec", Seq[StmD(d1); StmC(cmd1)]);;
printProg p;;
let (t, sigma1) = prgSem p;;
showState sigma1;;
let er = PreIncr(GetArrow("A", PostIncr(Val "i")));;
let c2 = Cmd(Upd(GetArrow("A", Val "i"), er));;
let s1 = StmC(Cmd(Upd(Val "i", N 0)));;
let eG = LT(Val "i", N 4);;
let cmd2 = For(s1,eG,sI,c2);;
let p2 = Prog("Vec", Seq[StmC cmd2]);;
printProg p2;;
let (t2, sigma2) = stmSem (Seq[StmC cmd2]) sigma1;;
showState sigma2;;
```

```
Program Vec{
   int array  A[5];
   for (var int i = 0;(i < 5);++i)
            A[i] = i;
...
   for (i = 0;(i < 4);++i)
            A[i] = (++A[(i++)]);
}
```

Detto i il valore di `i`, a il valore di `A[i]`, `A[i]=++A[i++]` è valutata nel modo seguente:

- Valuta `++A[i++]`: pre-incremento di e = `A[i++]`, cioè incrementa di 1 il valore di `A[i]` (che diventa $a+1$) e incrementa di 1 il valore di `i` (che diventa $i+1$).
- Assegna a `A[i]` (componente $i+1$-esima dell'array) il valore $a+1$.

```
let d1 = Array(Int, "A", 5);;
let sD = StmD(Var(Int, "i", N 0));;
let eG = LT(Val "i", N 5);;
let sI = StmC(Cmd(PreIncr(Val "i")));;
let c1 = Cmd(Upd(GetArrow("A", Val "i"), Val "i"));;
let cmd1 = For(sD,eG,sI,c1);;
let p = Prog("Vec", Seq[StmD(d1); StmC(cmd1)]);;
printProg p;;
let (t, sigma1) = prgSem p;;
showState sigma1;;
let er = PreIncr(GetArrow("A", PostIncr(Val "i")));;
let c2 = Cmd(Upd(GetArrow("A", Val "i"), er));;
let s1 = StmC(Cmd(Upd(Val "i", N 0)));;
let eG = LT(Val "i", N 4);;
let cmd2 = For(s1,eG,sI,c2);;
let p2 = Prog("Vec", Seq[StmC cmd2]);;
printProg p2;;
let (t2, sigma2) = stmSem (Seq[StmC cmd2]) sigma1;;
showState sigma2;;
```

```
Program Vec{
   int array  A[5];
   for (var int i = 0;(i < 5);++i)
            A[i] = i;
...
   for (i = 0;(i < 4);++i)
            A[i] = (++A[(i++)]);
}
```

```
Stack:
[i/(int,L5); A/(int[5],L0)]
Store:
[L0<-0,L1<-1,L2<-2,L3<-3,L4<-4,L5<-5]
```

`i = 5; A = [0,1,2,3,4]`

Detto i il valore di `i`, a il valore di `A[i]`, `A[i]=++A[i++]` è valutata nel modo seguente:

- Valuta `++A[i++]`: pre-incremento di e = `A[i++]`, cioè incrementa di 1 il valore di `A[i]` (che diventa $a+1$) e incrementa di 1 il valore di `i` (che diventa $i+1$).
- Assegna a `A[i]` (componente $i+1$-esima dell'array) il valore $a+1$.

```
let d1 = Array(Int, "A", 5);;
let sD = StmD(Var(Int, "i", N 0));;
let eG = LT(Val "i", N 5);;
let sI = StmC(Cmd(PreIncr(Val "i")));;
let c1 = Cmd(Upd(GetArrow("A", Val "i"), Val "i"));;
let cmd1 = For(sD,eG,sI,c1);;
let p = Prog("Vec", Seq[StmD(d1); StmC(cmd1)]);;
printProg p;;
let (t, sigma1) = prgSem p;;
showState sigma1;;
let er = PreIncr(GetArrow("A", PostIncr(Val "i")));;
let c2 = Cmd(Upd(GetArrow("A", Val "i"), er));;
let s1 = StmC(Cmd(Upd(Val "i", N 0)));;
let eG = LT(Val "i", N 4);;
let cmd2 = For(s1,eG,sI,c2);;
let p2 = Prog("Vec", Seq[StmC cmd2]);;
printProg p2;;
let (t2, sigma2) = stmSem (Seq[StmC cmd2]) sigma1;;
showState sigma2;;
```

```
Program Vec{
   int array  A[5];
   for (var int i = 0;(i < 5);++i)
            A[i] = i;
...
   for (i = 0;(i < 4);++i)
            A[i] = (++A[(i++)]);
}
```

```
Stack:
[i/(int,L5); A/(int[5],L0)]
Store:
[L0<-0,L1<-1,L2<-2,L3<-3,L4<-4,L5<-5]
```

`i = 5; A = [0,1,2,3,4]`

Detto i il valore di `i`, a il valore di `A[i]`, `A[i]=++A[i++]` è valutata nel modo seguente:

- Valuta `++A[i++]`: pre-incremento di e = `A[i++]`, cioè incrementa di 1 il valore di `A[i]` (che diventa $a+1$) e incrementa di 1 il valore di `i` (che diventa $i+1$).
- Assegna a `A[i]` (componente $i+1$-esima dell'array) il valore $a+1$.

```
let d1 = Array(Int, "A", 5);;
let sD = StmD(Var(Int, "i", N 0));;
let eG = LT(Val "i", N 5);;
let sI = StmC(Cmd(PreIncr(Val "i")));;
let c1 = Cmd(Upd(GetArrow("A", Val "i"), Val "i"));;
let cmd1 = For(sD,eG,sI,c1);;
let p = Prog("Vec", Seq[StmD(d1); StmC(cmd1)]);;
printProg p;;
let (t, sigma1) = prgSem p;;
showState sigma1;;
let er = PreIncr(GetArrow("A", PostIncr(Val "i")));;
let c2 = Cmd(Upd(GetArrow("A", Val "i"), er));;
let s1 = StmC(Cmd(Upd(Val "i", N 0)));;
let eG = LT(Val "i", N 4);;
let cmd2 = For(s1,eG,sI,c2);;
let p2 = Prog("Vec", Seq[StmC cmd2]);;
printProg p2;;
let (t2, sigma2) = stmSem (Seq[StmC cmd2]) sigma1;;
showState sigma2;;
```

Detto i il valore di `i`, a il valore di `A[i]`, `A[i]=++A[i++]` è valutata nel modo seguente:

- Valuta `++A[i++]`: pre-incremento di e = `A[i++]`, cioè incrementa di 1 il valore di `A[i]` (che diventa $a+1$) e incrementa di 1 il valore di `i` (che diventa $i+1$).
- Assegna a `A[i]` (componente $i+1$-esima dell'array) il valore $a+1$.

```
Program Vec{
   int array  A[5];
   for (var int i = 0;(i < 5);++i)
            A[i] = i;
...
   for (i = 0;(i < 4);++i)
            A[i] = (++A[(i++)]);
}
```

```
Stack:
[i/(int,L5); A/(int[5],L0)]
Store:
[L0<-0,L1<-1,L2<-2,L3<-3,L4<-4,L5<-5]
```

`i = 5; A = [0,1,2,3,4]`

```
Stack:
[i/(int,L5); A/(int[5],L0)]
Store:
[L0<-1,L1<-1,L2<-3,L3<-3,L4<-4,L5<-4]
```

`i = 4; A = [1,1,3,3,4]`