`nat` come sottotipo di `int`

Seminario di fine corso
Linguaggi di Programmazione e Laboratorio

Andrea Marino

Università di Pisa, Dipartimento di Matematica

17 Settembre 2021

Sommario

Sommario

Introduzione al problema

Durante il corso abbiamo visto un esempio di funzione parziale sugli interi, ossia non definita per alcuni valori.

Introduzione al problema

Durante il corso abbiamo visto un esempio di funzione parziale sugli interi, ossia non definita per alcuni valori. Sto parlando della funzione che calcola il fattoriale di un intero.

```
Program 5fattoriale_ricors{
    int n = 5;
    nat fattRic(value nat x){
          if (x == 0) return 1;
          if ((x == 0) == false) return (x * fattRic((x - 1)));

          }
    int nfatt = fattRic(Cast(nat, n));
    }

- : unit/2 = ()
#
```

Figura: *Output printProg*

$$fact(n) = \begin{cases} n! & \text{se } n \geq 0 \\ \bot & \text{se } n < 0 \end{cases}$$

Funzione matematica

```
let d1 = Var(Int, "n", N 5) in
let fp = FP(Value, Int, "x") in
let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
let espr = Times(Val "x", espaux) in
let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
let bl = BlockP(ED,SeqS(aux1,aux2)) in
let d2 = Pcd(Int, "fattRic", fp, bl) in
let d3 = Var(Int, "nfatt", Apply("fattRic", AP(Val "n"))) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("5fattoriale_ricors", Block(d,UnL ES)) in printProg prog
```

Figura: *Sintassi astratta*

Compatibilità e sottotipi

L'esempio precedente ci motiva ad introdurre un nuovo tipo, `nat`, per i numeri naturali.

Compatibilità e sottotipi

L'esempio precedente ci motiva ad introdurre un nuovo tipo, `nat`, per i numeri naturali. Small21 ha già un tipo per gli interi, quindi questo nuovo tipo dovrà essere introdotto come suo *sottotipo*.

Definizione

Un tipo `T` è compatibile con il tipo `S` se è possibile usare un valore di tipo `T` in un qualsiasi contesto in cui sarebbe richiesto un valore di tipo `S`.

Motivi per l'introduzione di `nat`

Sempre guidati dall'esempio precedente, osserviamo che l'introduzione di questo sottotipo insiste principalmente sull'aspetto di **Supporto alla correttezza**: uno degli scopi principali dei tipi è la verifica di proprietà statiche del programma.

Motivi per l'introduzione di `nat`

Sempre guidati dall'esempio precedente, osserviamo che l'introduzione di questo sottotipo insiste principalmente sull'aspetto di **Supporto alla correttezza**: uno degli scopi principali dei tipi è la verifica di proprietà statiche del programma.

Osservazione

I tipi possono comportarsi come *commenti effettivamente controllabili*: non solo essi esprimono le corrette modalità d'uso di un programma, ma ne vincolano anche l'uso.

Motivi per l'introduzione di `nat`

Sempre guidati dall'esempio precedente, osserviamo che l'introduzione di questo sottotipo insiste principalmente sull'aspetto di **Supporto alla correttezza**: uno degli scopi principali dei tipi è la verifica di proprietà statiche del programma.

Osservazione

I tipi possono comportarsi come *commenti effettivamente controllabili*: non solo essi esprimono le corrette modalità d'uso di un programma, ma ne vincolano anche l'uso.

Osservazione

Se una funzione che calcola il fattoriale avesse parametri formali di tipo `nat`, impedirebbe computazioni non terminanti, restituendo un errore nel caso in cui ricevesse in input un numero negativo

Sintassi

- Sintassi concreta

```
  ...
Simple ⟶ int | bool | nat
...
ExpA ⟶ ...| Cast(Simple,ExpA) | ExpA * ExpT |
```

Sintassi

- Sintassi concreta

```
  ...
Simple ⟶ int | bool | nat
...
ExpA ⟶ ...| Cast(Simple,ExpA) | ExpA * ExpT |
```

- Sintassi astratta

```
 Type ::= ...| [nat] | ...
...
Exp ::= ...| [cast] Type Exp | Exp [*] Exp | ...
```

Sintassi

- Sintassi concreta

```
  ...
Simple ⟶ int | bool | nat
...
ExpA ⟶ ...| Cast(Simple,ExpA) | ExpA * ExpT |
```

- Sintassi astratta

```
 Type ::= ...| [nat] | ...
...
Exp ::= ...| [cast] Type Exp | Exp [*] Exp | ...
```

L'operazione di prodotto $*$ non è strettamente necessaria, è stata introdotta solo perché utile per gli esempi trattati.

Nuova relazione tra tipi

Primo e secondo assioma del sottotipo:

$$[\text{Y0}]\frac{\mathtt{t}_1 = \mathtt{t}_2}{\mathtt{t}_1 \leq \mathtt{t}_2}$$

$$[\text{Y0'}]\frac{}{[\mathtt{nat}] \leq [\mathtt{int}]}$$

Nuova relazione tra tipi

Primo e secondo assioma del sottotipo:

$$[\text{Y0}]\frac{\mathtt{t}_1 = \mathtt{t}_2}{\mathtt{t}_1 \leq \mathtt{t}_2}$$

$$[\text{Y0'}]\frac{}{[\mathtt{nat}] \leq [\mathtt{int}]}$$

Questi due assiomi permettono di inferire ad un'espressione di tipo [nat] il tipo [int].
Ciò giustifica l'introduzione della nuova relazione di compatibilità tra tipi $\leq$, definita da: se t, t' sono tipi, t' $\leq$ t se e solo se t' = t o t' = [nat] e t = [int].

Nuova relazione tra tipi

Primo e secondo assioma del sottotipo:

$$[\text{Y0}]\frac{\mathtt{t}_1 = \mathtt{t}_2}{\mathtt{t}_1 \leq \mathtt{t}_2}$$

$$[\text{Y0'}]\frac{}{[\mathtt{nat}] \leq [\mathtt{int}]}$$

Questi due assiomi permettono di inferire ad un'espressione di tipo [nat] il tipo [int].
Ciò giustifica l'introduzione della nuova relazione di compatibilità tra tipi $\leq$, definita da: se t, t' sono tipi, t' $\leq$ t se e solo se t' = t o t' = [nat] e t = [int].

Inoltre aggiungiamo [nat] ai tipi semplici: $\mathtt{Simple} = \{[\mathtt{int}], [\mathtt{bool}], [\mathtt{nat}]\}$

AM21 è – essenzialmente – immutata.

Aggiunta all'interprete Ocaml la funzione `ycompatible` che verifica la compatibilità fra tipi:

```
let ysame t1 t2 = t1 = t2
   ;;

        (* ********* added by me ********* *)
let ycompatible t1 t2 =
   match (t1,t2) with
   | (Nat, Int) -> true
   | (a,b) -> ysame a b
   ;;
```

Figura: *Funzioni* `ysame` *e* `ycompatible` *a confronto.*

AM21 è – essenzialmente – immutata.

Aggiunta all'interprete Ocaml la funzione `ycompatible` che verifica la compatibilità fra tipi:

```
let ysame t1 t2 = t1 = t2
   ;;

        (* ********* added by me ********* *)
let ycompatible t1 t2 =
   match (t1,t2) with
   | (Nat, Int) -> true
   | (a,b) -> ysame a b
   ;;
```

Figura: *Funzioni* `ysame` *e* `ycompatible` *a confronto.*

Molte delle modifiche apportate all'interprete di Small21 (dichiarazione, assegnamento, trasmissione di parametri) si riducono al sostituire `ysame` con `ycompatible`.

Sommario

(modifica delle) Regole per DCL: Sistema Y

$$[\text{Y1'}]\ \frac{\langle \texttt{e},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{t'}, \mathtt{Y}_\rho) \qquad \texttt{t} \in \texttt{Simple} \qquad \texttt{t'} \leq \texttt{t} \qquad \mathtt{Y}_\rho => \rho :: \mathtt{Y}'_\rho] \qquad \rho(\texttt{I}) = \bot \qquad > [\texttt{I}/\texttt{t}] \circ \rho :: \mathtt{Y}'_\rho] = \mathtt{Y}''_\rho}{\langle [\texttt{const}]\ \texttt{t}\ \texttt{I}\ \texttt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} \left([\texttt{void}], \mathtt{Y}''_\rho\right)}$$

$$[\text{Y2'}]\ \frac{\langle \texttt{e},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{t'}, \mathtt{Y}_\rho) \qquad \texttt{t} \in \texttt{Simple} \qquad (\texttt{t'} \leq \texttt{t}\ \texttt{or}\ \texttt{t'}=[\texttt{unit}]) \qquad \mathtt{Y}_\rho\big|_0(\texttt{I}) = \bot}{\langle [\texttt{var}]\ \texttt{t}\ \texttt{I}\ \texttt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\texttt{void}], [\texttt{I}/[\texttt{mut}]\ \texttt{t}] \otimes \mathtt{Y}_\rho)}$$

Errori di tipo:

$$[\text{E2'}]\ \frac{\langle \texttt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{t'}, \mathtt{Y}_\rho) \qquad \texttt{t'} \not\leq \texttt{t}}{\langle [\texttt{const}]\ \texttt{t}\ \texttt{I}\ \texttt{e}, \mathtt{Y}_\rho\rangle \to ([\texttt{terr}], \mathtt{Y}_\rho)} \qquad [\text{E5'}]\ \frac{\langle \texttt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{t'}, \mathtt{Y}_\rho) \qquad \texttt{t'} \not\leq \texttt{t} \qquad \texttt{t'} \neq [\texttt{unit}]}{\langle [\texttt{var}\ \texttt{t}\ \texttt{I}\ \texttt{e}, \mathtt{Y}_\rho\rangle \to ([\texttt{terr}], \mathtt{Y}_\rho)}$$

(modifica delle) Regole per DCL: Sistema Y

$$[\text{Y1'}]\ \frac{\langle \texttt{e},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{t'}, \mathtt{Y}_\rho) \qquad \texttt{t} \in \texttt{Simple} \qquad \texttt{t'} \leq \texttt{t} \qquad \mathtt{Y}_\rho \Rightarrow \rho :: \mathtt{Y}'_\rho] \qquad \rho(\mathtt{I}) = \bot \qquad \Rightarrow [\mathtt{I}/\texttt{t}] \circ \rho :: \mathtt{Y}'_\rho] = \mathtt{Y}''_\rho}{\langle [\texttt{const}]\ \texttt{t}\ \mathtt{I}\ \texttt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} \left([\texttt{void}], \mathtt{Y}''_\rho\right)}$$

$$[\text{Y2'}]\ \frac{\langle \texttt{e},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{t'}, \mathtt{Y}_\rho) \qquad \texttt{t} \in \texttt{Simple} \qquad (\texttt{t'} \leq \texttt{t} \text{ or } \texttt{t'} = [\texttt{unit}]) \qquad \mathtt{Y}_\rho\big|_0(\mathtt{I}) = \bot}{\langle [\texttt{var}]\ \texttt{t}\ \mathtt{I}\ \texttt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} \left([\texttt{void}], [\mathtt{I}/[\texttt{mut}]\ \texttt{t}] \otimes \mathtt{Y}_\rho\right)}$$

Errori di tipo:

$$[\text{E2'}]\ \frac{\langle \texttt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{t'}, \mathtt{Y}_\rho) \qquad \texttt{t'} \not\leq \texttt{t}}{\langle [\texttt{const}]\ \texttt{t}\ \mathtt{I}\ \texttt{e}, \mathtt{Y}_\rho\rangle \to ([\texttt{terr}], \mathtt{Y}_\rho)} \qquad [\text{E5'}]\ \frac{\langle \texttt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{t'}, \mathtt{Y}_\rho) \qquad \texttt{t'} \not\leq \texttt{t} \qquad \texttt{t'} \neq [\texttt{unit}]}{\langle [\texttt{var}\ \texttt{t}\ \mathtt{I}\ \texttt{e}, \mathtt{Y}_\rho\rangle \to ([\texttt{terr}], \mathtt{Y}_\rho)}$$

Notazione: $\texttt{Simple} = \{[\texttt{int}], [\texttt{nat}], [\texttt{bool}]\}$. Relazione $\leq$ di compatibilità tra tipi: $\texttt{t'} \leq \texttt{t}$.

Quando $\texttt{t'}, \texttt{t}$ sono tipi, $(\texttt{t'} \leq \texttt{t}) = \texttt{true}$ s. se $\texttt{t'} = \texttt{t}$ or $\texttt{t'} < \texttt{t}$

(modifica delle) Regole per DCL: Sistema $\mathtt{SEM}_{\mathtt{DCL}}$

$$[\text{D1'}]\ \frac{\langle \mathtt{e}, (\Delta, \mu)\rangle \to \lfloor \mathtt{t_e}, \mathtt{v}\,(\Delta, \mu_\mathtt{e})\rfloor \qquad \mathtt{t} \in \mathtt{Simple} \qquad \mathtt{t_e} \leq \mathtt{t} \qquad \Delta\big|_0(\mathtt{I}) = \bot \qquad [\mathtt{I}/(\mathtt{t_e}, \mathtt{v})] \otimes \Delta = \Delta_\mathtt{I}}{\langle [\mathtt{const}]\ \mathtt{t}\ \mathtt{I}\ \mathtt{e}, (\Delta, \mu)\rangle \to ([\mathtt{void}], (\Delta_\mathtt{I}, \mu_\mathtt{e}))}$$

$$[\text{D2''}]\ \frac{\langle \mathtt{e}, (\Delta, \mu_\mathtt{e})\rangle \to \lfloor \mathtt{t_e}, \mathtt{v}\,(\Delta, \mu)\rfloor \qquad \mathtt{t} \in \mathtt{Simple} \qquad \mathtt{t_e} \leq \mathtt{t} \qquad \Delta\big|_0(\mathtt{I}) = \bot \qquad \triangleright\,(\mu_\mathtt{e}, \mathtt{t}, 1) = (\mathtt{loc_t}, \mu_\mathtt{a}) \qquad \mu_\mathtt{a}\,(\mathtt{loc_t} \leftarrow \mathtt{v}) = \mu'_\mathtt{a} \qquad [\mathtt{I}/(\mathtt{Mut}\ \mathtt{t_e}, \mathtt{v})] \otimes \Delta = \Delta_\mathtt{I}}{\langle [\mathtt{var}]\ \mathtt{t}\ \mathtt{I}\ \mathtt{e}, (\Delta, \mu)\rangle \to ([\mathtt{void}], (\Delta_\mathtt{I}, \mu'_\mathtt{a}))}$$

Notazione: $\mathtt{Simple} = \{[\mathtt{int}], [\mathtt{nat}], [\mathtt{bool}]\}$. Relazione $\leq$ di compatibilità tra tipi: $\mathtt{t'} \leq \mathtt{t}$.

Quando $\mathtt{t'}$, $\mathtt{t}$ sono tipi, $(\mathtt{t'} \leq \mathtt{t}) = \mathtt{true}$ s. se $\mathtt{t'} = \mathtt{t}$ or $\mathtt{t'} < \mathtt{t}$

Sistema Y: (modifica delle) Regole per call e apply

$$[\text{Y25'}]\ \frac{\mathtt{Y}_\rho(\mathtt{ide}) = \mathtt{[abs][void][::]t'} \quad \mathtt{aps} = \mathtt{[AP]\ exp} \quad \langle \mathtt{exp},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{ta}, \mathtt{Y}_\rho) \quad \mathtt{ta} \leq \mathtt{t'}}{\langle \mathtt{[call]\ ide\ aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{[void]}, \mathtt{Y}_\rho)}$$

$$[\text{Y26'}]\ \frac{\mathtt{Y}_\rho(\mathtt{ide}) = \mathtt{[abs]t[::]t'} \quad \mathtt{aps} = \mathtt{[AP]\ exp} \quad \langle \mathtt{exp},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{ta}, \mathtt{Y}_\rho) \quad \mathtt{t} \in \mathtt{Simple} \quad \mathtt{ta} \leq \mathtt{t'}}{\langle \mathtt{[apply]\ ide\ aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{[void]}, \mathtt{Y}_\rho)}$$

Errori di tipo:

$$[\text{E47}']\ \frac{\mathtt{Y}_\rho(\mathtt{ide}) = \mathtt{[abs][void][::]t'} \quad \mathtt{aps} = \mathtt{[AP]\ exp} \quad \langle \mathtt{exp},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{ta}, \mathtt{Y}_\rho) \quad \mathtt{ta} \not\leq \mathtt{t'}}{\langle \mathtt{[call]\ ide\ aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{[terr]}, \mathtt{Y}_\rho)}$$

$$[\text{E34}']\ \frac{\mathtt{Y}_\rho(\mathtt{ide}) = \mathtt{[abs]t[::]t'} \quad \mathtt{aps} = \mathtt{[AP]\ exp} \quad \langle \mathtt{exp},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{ta}, \mathtt{Y}_\rho) \quad \mathtt{t} \in \mathtt{Simple} \quad \mathtt{ta} \not\leq \mathtt{t'}}{\langle \mathtt{[apply]\ ide\ aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{[terr]}, \mathtt{Y}_\rho)}$$

Notazione: $\mathtt{Simple} = \{\mathtt{[int]}, \mathtt{[nat]}, \mathtt{[bool]}\}$. Relazione $\leq$ di compatibilità tra tipi: $\mathtt{t'} \leq \mathtt{t}$.
Quando $\mathtt{t'}, \mathtt{t}$ sono tipi, $(\mathtt{t'} \leq \mathtt{t}) = \mathtt{true}$ s. se $\mathtt{t'} = \mathtt{t}$ or $\mathtt{t'} < \mathtt{t}$

Sistema Y: (modifica delle) Regole di trasmissione dei parametri

$$[\text{Y35'}]\ \frac{\begin{array}{c}\mathtt{fps} = [\mathtt{fp}][\mathtt{value}]\,\mathtt{t}\,\mathtt{ide} \\ \mathtt{Y}_\rho\big|_0(\mathtt{I}) = \bot \\ \mathtt{aps} = [\mathtt{ap}]\mathtt{exp} \\ \langle \mathtt{exp},\, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{ta}, \mathtt{Y}_\rho) \\ \mathtt{ta} \leq \mathtt{t} \qquad \mathtt{t} \in \mathtt{Simple}\end{array}}{\langle \mathtt{fps} \lhd \mathtt{aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{void}], \mathtt{Y}_\rho)}$$

$$[\text{Y36'}]\ \frac{\begin{array}{c}\mathtt{fps} = [\mathtt{fp}][\mathtt{ref}]\,\mathtt{t}\,\mathtt{ide} \\ \mathtt{Y}_\rho\big|_0(\mathtt{I}) = \bot \\ \mathtt{aps} = [\mathtt{ap}]\,\mathtt{exp} \\ \langle \mathtt{exp},\, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} ([\mathtt{Mut}]\,\mathtt{ta}, \mathtt{Y}_\rho) \\ \mathtt{ta} \leq \mathtt{t} \qquad \mathtt{t} \in \mathtt{Simple}\end{array}}{\langle \mathtt{fps} \lhd \mathtt{aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{void}], \mathtt{Y}_\rho)}$$

Sistema Y: (modifica delle) Regole di trasmissione dei parametri

$$[\text{Y35'}]\ \frac{\begin{array}{c}\texttt{fps} = [\texttt{fp}][\texttt{value}]\,\texttt{t}\,\texttt{ide} \\ \mathtt{Y}_\rho\big|_0(\mathtt{I}) = \bot \\ \texttt{aps} = [\texttt{ap}]\texttt{exp} \\ \langle \texttt{exp},\, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\texttt{ta}, \mathtt{Y}_\rho) \\ \texttt{ta} \leq \texttt{t} \qquad \texttt{t} \in \texttt{Simple}\end{array}}{\langle \texttt{fps} \lhd \texttt{aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\texttt{void}], \mathtt{Y}_\rho)}$$

$$[\text{Y36'}]\ \frac{\begin{array}{c}\texttt{fps} = [\texttt{fp}][\texttt{ref}]\,\texttt{t}\,\texttt{ide} \\ \mathtt{Y}_\rho\big|_0(\mathtt{I}) = \bot \\ \texttt{aps} = [\texttt{ap}]\,\texttt{exp} \\ \langle \texttt{exp},\, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} ([\texttt{Mut}]\,\texttt{ta}, \mathtt{Y}_\rho) \\ \texttt{ta} \leq \texttt{t} \qquad \texttt{t} \in \texttt{Simple}\end{array}}{\langle \texttt{fps} \lhd \texttt{aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\texttt{void}], \mathtt{Y}_\rho)}$$

Errori di tipo:

$$[\text{E61.1'}]\ \frac{\begin{array}{c}\ldots \\ \texttt{ta} \not\leq \texttt{t}\end{array}}{\langle \texttt{fps} \lhd \texttt{aps}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\texttt{terr}], \mathtt{Y}_\rho)}$$

Notazione: $\texttt{Simple} = \{[\texttt{int}], [\texttt{nat}], [\texttt{bool}]\}$. Relazione $\leq$ di compatibilità tra tipi: $\texttt{t'} \leq \texttt{t}$.
Quando $\texttt{t'}, \texttt{t}$ sono tipi, $(\texttt{t'} \leq \texttt{t}) = \texttt{true}$ s. se $\texttt{t'} = \texttt{t}$ or $\texttt{t'} < \texttt{t}$

(modifica delle) Regole per trasmissione dei parametri: Sistema TR1

$$\mathtt{fps} = [\mathtt{fp}][\mathtt{value}]\,\mathtt{t}\,\mathtt{I}$$

$$\mathtt{aps} = [\mathtt{ap}]\,\mathtt{exp}$$

$$\langle \mathtt{exp}, (\Delta, \mu)\rangle \rightarrow \lfloor \mathtt{ta}, \mathtt{va}, (\Delta_1, \mu_1)\rfloor$$

$$\mathtt{t} \in \mathtt{Simple} \qquad \mathtt{ta} \leq \mathtt{t} \qquad \Delta_{\mathtt{C}}\big|_0(\mathtt{I}) = \perp$$

$$\triangleright\,(\mu_1, \mathtt{t}, 1) = (\mathtt{loc}_{\mathtt{t}}, \mu_2)$$

$$\mu_2[\mathtt{loc}_{\mathtt{t}} \leftarrow \mathtt{va}] = \mu_3$$

$$[\mathtt{I}/\,([\mathtt{Mut}]\,\mathtt{t}, \mathtt{loc}_{\mathtt{t}})] \otimes \Delta_{\mathtt{C}} = \Delta_{\mathtt{C}}^{\mathtt{F}}$$

$$[\mathtt{S16}]\,\frac{(\Delta_{\mathtt{C}}^{\mathtt{F}}, \mu_3) = \sigma_{\mathtt{r}} \qquad [] = \mathtt{epi}_{\mathtt{r}}}{\langle \mathtt{fps} \lhd \mathtt{aps}, (\Delta, \Delta_{\mathtt{C}}, \mu)\rangle \rightarrow_{\mathtt{TR1}} (\sigma_{\mathtt{r}}, \mathtt{epi}_{\mathtt{r}})}$$

Notazione: $\mathtt{Simple} = \{[\mathtt{int}], [\mathtt{nat}], [\mathtt{bool}]\}$. Relazione $\leq$ di compatibilità tra tipi: $\mathtt{t'} \leq \mathtt{t}$.

Quando $\mathtt{t'}, \mathtt{t}$ sono tipi, $(\mathtt{t'} \leq \mathtt{t}) = \mathtt{true}$ s. se $\mathtt{t'} = \mathtt{t}$ or $\mathtt{t'} < \mathtt{t}$

(modifica delle) Regole per trasmissione dei parametri: Sistema TR1

$$\mathtt{fps} = [\mathtt{fp}][\mathtt{Ref}]\,\mathtt{t}\,\mathtt{I}$$

$$\mathtt{aps} = [\mathtt{ap}]\,\mathtt{exp}$$

$$\langle \mathtt{exp},\, (\Delta, \mu)\rangle \rightarrow_{\mathtt{DEN}} \lfloor [\mathtt{Mut}]\mathtt{ta}, \mathtt{loca}, (\Delta_1, \mu_1)\rfloor$$

$$\mathtt{t} \in \mathtt{Simple} \qquad \mathtt{ta} \leq \mathtt{t} \qquad \Delta_{\mathtt{C}}\big|_0(\mathtt{I}) = \perp$$

$$[\mathtt{I}/\,([\mathtt{Mut}]\,\mathtt{ta}, \mathtt{loca})] \otimes \Delta_{\mathtt{C}} = \Delta_{\mathtt{C}}^{\mathtt{F}}$$

$$[\mathtt{S17}]\ \frac{\left(\Delta_{\mathtt{C}}^{\mathtt{F}}, \mu_3\right) = \sigma_{\mathtt{r}} \qquad [] = \mathtt{epi}_{\mathtt{r}}}{\langle \mathtt{fps} \lhd \mathtt{aps}, (\Delta, \Delta_{\mathtt{C}}, \mu)\rangle \rightarrow_{\mathtt{TR1}} (\sigma_{\mathtt{r}}, \mathtt{epi}_{\mathtt{r}})}$$

Notazione: $\mathtt{Simple} = \{[\mathtt{int}], [\mathtt{nat}], [\mathtt{bool}]\}$. Relazione $\leq$ di compatibilità tra tipi: $\mathtt{t'} \leq \mathtt{t}$.

Quando $\mathtt{t'}, \mathtt{t}$ sono tipi, $(\mathtt{t'} \leq \mathtt{t}) = \mathtt{true}$ s. se $\mathtt{t'} = \mathtt{t}$ or $\mathtt{t'} < \mathtt{t}$

(modifica delle) Regole di assegnamento

Sistema Y:

$$[\text{Y15'}]\ \frac{\langle \mathtt{e}_1,\ \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{t}_1, \mathtt{Y}_\rho) \quad \langle \mathtt{e}_\mathtt{r},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{r}, \mathtt{Y}_\rho) \quad \mathtt{t}_1 = [\mathtt{mut}]\,\mathtt{t} \quad \mathtt{t}_\mathtt{r} \leq \mathtt{t} \quad \mathtt{t} \in \mathtt{Simple}}{\langle \mathtt{e}_1[=]\,\mathtt{e}_\mathtt{r}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)}$$

Errori di tipo:

$$[\text{E19'}]\ \frac{\langle \mathtt{e}_1,\ \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} (\mathtt{t}_1, \mathtt{Y}_\rho) \quad \langle \mathtt{e}_\mathtt{r},\ \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{r}, \mathtt{Y}_\rho) \quad \mathtt{t}_1 = [\mathtt{Mut}]\,\mathtt{t} \quad \mathtt{t}_\mathtt{r} \not\leq \mathtt{t}}{\langle \mathtt{e}_1[=]\,\mathtt{e}_\mathtt{r}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

Regole di inferenza $\mathtt{SEM}_{\mathtt{STM}}$:

$$[\text{S1'}]\ \frac{\langle \mathtt{e}_1,\ \sigma\rangle \to \lfloor \mathtt{t}_\mathtt{r},\ \mathtt{v}_\mathtt{r},\ \sigma_\mathtt{r}\rfloor \quad \langle \mathtt{e}_\mathtt{r},\ \sigma_\mathtt{r}\rangle \to_{\mathtt{DEN}} \lfloor \mathtt{t}_1,\ \mathtt{Loc}_\mathtt{t},\ \sigma_1\rfloor \quad \mathtt{t}_1 = [\mathtt{mut}]\,\mathtt{t} \quad \mathtt{t}_\mathtt{r} \leq \mathtt{t} \quad \mathtt{t} \in \mathtt{Simple} \quad \sigma_1 = (\Delta_1, \mu_1) \quad \mu_1\,(\mathtt{Loc}_t \leftarrow \mathtt{v}_\mathtt{r}) = \mu_\mathtt{F}}{\langle \mathtt{e}_1[=]\,\mathtt{e}_\mathtt{r}, \sigma\rangle \to ([\mathtt{void}], (\Delta_1, \mu_\mathtt{F}))}$$

(modifica della) regola per Return

Sistema dei tipi e errori di tipo: Non servono modifiche.

Notazione: `Simple`$= \{[\texttt{int}], [\texttt{nat}], [\texttt{bool}]\}$. Relazione $\leq$ di compatibilità tra tipi: `t'` $\leq$ `t`.

Quando `t'`, `t` sono tipi, (`t'` $\leq$ `t`) $=$ `true` s. se `t'` $=$ `t` `or` `t'` $<$ `t`

(modifica della) regola per Return

Sistema dei tipi e errori di tipo: Non servono modifiche.

Regole di inferenza $\mathtt{SEM}_{\mathtt{STM}}$:

$$[\text{S18'}]\ \frac{\begin{array}{c}\langle \mathtt{e},\sigma\rangle \to \lfloor \mathtt{t_e}, \mathtt{v_e}\,(\Delta_\mathtt{e}, \mu_\mathtt{e})\rfloor \\ \Delta_\mathtt{e} = \mathtt{ar} + \Delta_\mathtt{r} \\ \mathtt{ar} = \{\mathtt{id}, \mathtt{ls}, \mathtt{fr}, \mathtt{cnt}, \mathtt{v}\} \\ \mathtt{id} \in \mathtt{Ide} \qquad \Delta_\mathtt{e} = (\mathtt{t_r}, \mathtt{c_r}) \\ \mathtt{t_r} = [\mathtt{abs}]\,\mathtt{t}[::]\mathtt{t}' \\ (\mathtt{t_e} \leq \mathtt{t}) \vee (\mathtt{t} = [\mathtt{void}] \wedge \mathtt{t_e} = [\mathtt{unit}]) \\ \{\mathtt{id}, \mathtt{ls}, \mathtt{fr}, [], \mathtt{v_e}\} = \mathtt{ar1} \\ (\mathtt{ar1} + \Delta_\mathtt{r}, \mu_\mathtt{e}) = \sigma_\mathtt{F}\end{array}}{\langle [\mathtt{return}]\,\mathtt{e}, \sigma\rangle \to ([\mathtt{void}], \sigma_\mathtt{F})}$$

Notazione: $\mathtt{Simple} = \{[\mathtt{int}], [\mathtt{nat}], [\mathtt{bool}]\}$. Relazione $\leq$ di compatibilità tra tipi: $\mathtt{t'} \leq \mathtt{t}$.

Quando $\mathtt{t'}, \mathtt{t}$ sono tipi, $(\mathtt{t'} \leq \mathtt{t}) = \mathtt{true}$ s. se $\mathtt{t'} = \mathtt{t}$ or $\mathtt{t'} < \mathtt{t}$

Sommario

(modifica delle) Operazioni su Array (1)

Sistema Y:

$$[\text{Y15'}]\ \frac{\mathtt{Y}_\rho(\mathtt{I}) = [\mathtt{arr}]\,([\mathtt{Mut}]\,\mathtt{t})\ \ \mathtt{N} \qquad \langle \mathtt{e},\, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{t_e}, \mathtt{Y}_\rho) \qquad \mathtt{t_e} \leq [\mathtt{int}] \qquad \mathtt{DynamicOutBoundCheck}}{\langle \mathtt{I}\,[\uparrow 1]\,\mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} ([\mathtt{Mut}]\,\mathtt{t}, \mathtt{Y}_\rho)}$$

$$[\text{Y16'}]\ \frac{\mathtt{Y}_\rho(\mathtt{I}) = [\mathtt{arr}]\,([\mathtt{Mut}]\,\mathtt{t})\ \ \mathtt{N} \qquad \langle \mathtt{e},\, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{t_e}, \mathtt{Y}_\rho) \qquad \mathtt{t_e} \leq [\mathtt{int}] \qquad \mathtt{DynamicOutBoundCheck}}{\langle \mathtt{I}\,[\uparrow 1]\,\mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)}$$

Errori di tipo:

$$[\text{E30'}]\ \frac{\ldots \qquad \langle \mathtt{e},\, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{t_e}, \mathtt{Y}_\rho) \qquad \mathtt{t_e} \not\leq [\mathtt{int}]}{\langle \mathtt{I}\,[\uparrow 1]\,\mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DY}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

$$[\text{E30.1'}]\ \frac{\ldots \qquad \langle \mathtt{e},\, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} (\mathtt{t_e}, \mathtt{Y}_\rho) \qquad \mathtt{t_e} \not\leq [\mathtt{int}]}{\langle \mathtt{I}\,[\uparrow 1]\,\mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

(modifica delle) Operazioni su Array (2)

Regole di inferenza $\mathtt{SEM}_{\mathtt{EXP}}$:

$$\Delta(\mathtt{I}) = (\mathtt{t}, \mathtt{u}) \qquad \mathtt{u} = \mathtt{Loc}\ \mathtt{k}$$
$$\mathtt{t} = [\mathtt{arr}]\,([\mathtt{Mut}]\,\mathtt{t_r})\ \ \mathtt{N}$$
$$\langle \mathtt{e}, (\Delta, \mu)\rangle \to \lfloor \mathtt{t_e}, \mathtt{N_1}, \sigma_\mathtt{r} \rfloor$$
$$\mathtt{t_e} \leq [\mathtt{int}] \qquad \mathtt{N_1} \geq 0 \qquad \mathtt{N_1} < \mathtt{N}$$
$$[\mathrm{X7'}]\ \frac{\mathtt{t_r} \in \mathtt{Simple} \qquad \mathtt{Loc}\,(\mathtt{k} + \mathtt{N_1}) = \mathtt{Loc}'}{\langle \mathtt{I}\,[\uparrow 1]\,\mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DEXP}} \lfloor [\mathtt{Mut}]\,\mathtt{t_r}, \mathtt{Loc}', \sigma_\mathtt{r} \rfloor}$$

(modifica delle) Operazioni su Array (2)

Regole di inferenza $\mathtt{SEM}_{\mathtt{EXP}}$:

$$\Delta(\mathtt{I}) = (\mathtt{t}, \mathtt{u}) \qquad \mathtt{u} = \mathtt{Loc}\ \mathtt{k}$$
$$\mathtt{t} = [\mathtt{arr}]\,([\mathtt{Mut}]\,\mathtt{t_r})\ \ \mathtt{N}$$
$$\langle \mathtt{e}, (\Delta, \mu)\rangle \to \lfloor \mathtt{t_e}, \mathtt{N}_1, \sigma_\mathtt{r} \rfloor$$
$$\mathtt{t_e} \leq [\mathtt{int}] \qquad \mathtt{N}_1 \geq 0 \qquad \mathtt{N}_1 < \mathtt{N}$$
$$[\mathrm{X7'}]\ \frac{\mathtt{t_r} \in \mathtt{Simple} \qquad \mathtt{Loc}\,(\mathtt{k} + \mathtt{N}_1) = \mathtt{Loc}'}{\langle \mathtt{I}\,[\uparrow 1]\,\mathtt{e}, \mathtt{Y}_\rho\rangle \to_{\mathtt{DEXP}} \lfloor [\mathtt{Mut}]\,\mathtt{t_r}, \mathtt{Loc}', \sigma_\mathtt{r} \rfloor}$$

Osservazione

Interessante modifica: rendere [nat] il tipo indice degli Array di Small21. Ciò avrebbe reso innecessario `DynamicOutBoundCheck`, ma non avrebbe reso statico il controllo dell'ammissibilità degli indici array (se si usa `Cast` per fare tale controllo sul tipo (cfr.)).

Espressione Cast

Sintassi: `Cast(t, exp)`

Espressione Cast

Sintassi: `Cast`(`t`, `exp`)

Valuta `exp`, ne restituisce il valore e restituisce `t` $\in \{$[`nat`], [`int`]$\}$ come tipo (del valore) di `exp`, se compatibile.

Espressione Cast

Sintassi: $\texttt{Cast}(\texttt{t}, \texttt{exp})$

Valuta exp, ne restituisce il valore e restituisce $\texttt{t} \in \{[\texttt{nat}], [\texttt{int}]\}$ come tipo (del valore) di exp, se compatibile.

Sistema Y:

$$[\text{Y74.1}]\ \frac{\begin{array}{c}\texttt{t} = [\texttt{int}] \\ \langle \texttt{e},\, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{Y}} (\texttt{t}_\texttt{e}, \texttt{Y}_\rho) \\ \texttt{t}_\texttt{e} = [\texttt{int}] \text{ or } \texttt{t}_\texttt{e} = [\texttt{nat}]\end{array}}{\langle [\texttt{Cast}]\,\texttt{t}\ \texttt{e}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{Y}} ([\texttt{int}], \texttt{Y}_\rho)} \qquad [\text{Y74.2}]\ \frac{\begin{array}{c}\texttt{t} = [\texttt{nat}] \\ \langle \texttt{e},\, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{Y}} (\texttt{t}_\texttt{e}, \texttt{Y}_\rho) \\ \texttt{t}_\texttt{e} = [\texttt{int}] \text{ or } \texttt{t}_\texttt{e} = [\texttt{nat}] \\ \texttt{DynamicPositivityCheck}\end{array}}{\langle [\texttt{Cast}]\,\texttt{t}\ \texttt{e}, \texttt{Y}_\rho \rangle \rightarrow_{\texttt{Y}} ([\texttt{nat}], \texttt{Y}_\rho)}$$

DynamicPositivityCheck: Controllo a runtime di compatibilità del valore dell'espressione con il tipo [nat] (i.e. non negatività).

Espressione Cast

Sintassi: `Cast`$(\mathtt{t}, \mathtt{exp})$

Valuta exp, ne restituisce il valore e restituisce $\mathtt{t} \in \{[\mathtt{nat}], [\mathtt{int}]\}$ come tipo (del valore) di exp, se compatibile.

Errori di tipo:

$$[\text{E800}]\ \frac{\mathtt{t} \notin \{[\mathtt{int}], [\mathtt{nat}]\}}{\langle [\mathtt{Cast}]\,\mathtt{t}\ \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)} \qquad [\text{E801}]\ \frac{\begin{array}{c}\ldots \\ \langle \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t_e}, \mathtt{Y}_\rho) \\ \mathtt{t_e} \notin \{[\mathtt{int}], [\mathtt{nat}]\}\end{array}}{\langle [\mathtt{Cast}]\,\mathtt{t}\ \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

$$\ldots$$

$$[\text{E802}]\ \frac{\mathtt{DynamicPositivityCheck:Failure}}{\langle [\mathtt{Cast}]\,\mathtt{t}\ \mathtt{e}, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

Espressione Cast

Sintassi: `Cast`$(\mathtt{t}, \mathtt{exp})$

Valuta exp, ne restituisce il valore e restituisce $\mathtt{t} \in \{[\mathtt{nat}], [\mathtt{int}]\}$ come tipo (del valore) di exp, se compatibile.

Regole di inferenza $\mathrm{SEM}_{\mathrm{EXP}}$:

$$[\text{X10.1}]\ \frac{\begin{array}{c}\mathtt{t} = [\mathtt{int}] \\ \langle \mathtt{e}, \sigma \rangle \to \lfloor \mathtt{t_e}, \mathtt{v_e}, \sigma_\mathtt{e} \rfloor \\ (\mathtt{t_e} = [\mathtt{int}]\ \mathtt{or}\ \mathtt{t_e} = [\mathtt{nat}])\end{array}}{\langle [\mathtt{Cast}]\,\mathtt{t}\ \mathtt{e}, \sigma \rangle \to \lfloor [\mathtt{int}], \mathtt{v_e}, \sigma_\mathtt{e} \rfloor}$$

$$[\text{X10.2}]\ \frac{\begin{array}{c}\mathtt{t} = [\mathtt{nat}] \\ \langle \mathtt{e}, \sigma \rangle \to \lfloor \mathtt{t_e}, \mathtt{v_e}, \sigma_\mathtt{e} \rfloor \\ \mathtt{t_e} = [\mathtt{nat}]\ \mathtt{or}\ \ (\mathtt{t_e} = [\mathtt{int}]\ \mathtt{and}\ \mathtt{v_e} \geq 0)\end{array}}{\langle [\mathtt{Cast}]\,\mathtt{t}\ \mathtt{e}, \sigma \rangle \to \lfloor [\mathtt{nat}], \mathtt{v_e}, \sigma_\mathtt{e} \rfloor}$$

Espressione Cast

Sintassi: `Cast(t, exp)`

Valuta `exp`, ne restituisce il valore e restituisce $t \in \{[\texttt{nat}], [\texttt{int}]\}$ come tipo (del valore) di `exp`, se compatibile.

Modifica della funzione semantica `expSem`:

```
| Cast(t,exp)              (* ************* added by me ************* *)
   -> (match t with
          | Int ->
             (match expSem exp (sk,mu) with
                | (te, Ival n1, sge) when ((ysame te Int) || (ysame te Nat))
                   -> (Int, Ival n1, sge)
                | _
                   -> raise(TypeErrorE("E801: expSem: - ", exp)))
          | Nat ->
             (match expSem exp (sk,mu) with
                | (Nat, Ival n1, sge)
                   -> (Nat, Ival n1, sge)
                | (Int, Ival n1, sge) when (n1 >= 0)
                   -> (Nat, Ival n1, sge)
                | _
                   -> raise(TypeErrorE("E802: expSem: -",exp)))
          | _ -> raise(TypeErrorE("E800: expSem: - ",exp)))
```

(modifica alla) Espressione num

Modificata l'espressione num che restituisce un valore numerico:

Sistema Y:

$$[\text{Y8.1}]\ \frac{\mathtt{N} >= 0}{\langle [\mathtt{num}]\,\mathtt{N}, \mathtt{Y}_\rho\rangle \rightarrow_{\mathtt{Y}} ([\mathtt{nat}], \mathtt{Y}_\rho)} \qquad [\text{Y8.2}]\ \frac{\mathtt{N} < 0}{\langle [\mathtt{num}]\,\mathtt{N}, \mathtt{Y}_\rho\rangle \rightarrow_{\mathtt{Y}} ([\mathtt{int}], \mathtt{Y}_\rho)}$$

Regole di inferenza $\mathtt{SEM}_{\mathtt{EXP}}$:

$$[\text{X1.1}]\ \frac{\mathtt{N} >= 0}{\langle [\mathtt{num}]\,\mathtt{N}, \sigma\rangle \rightarrow \lfloor [\mathtt{nat}], \mathtt{N}, \sigma\rfloor} \qquad [\text{X1.2}]\ \frac{\mathtt{N} < 0}{\langle [\mathtt{num}]\,\mathtt{N}, \sigma\rangle \rightarrow \lfloor [\mathtt{int}], \mathtt{N}, \sigma\rfloor}$$

Funzione semantica expSem:

```
| N n ->          (* ********** changed by me ********** *)
    if n<0 then (Int,Ival n,(sk,mu))
    else  (Nat, Ival n, (sk,mu))
```

Espressioni aritmetiche e relazionali: problemi

L'introduzione del sottotipo [`nat`] di [`int`] ci obbliga ad apportare delle modifiche alle espressioni binarie di Small21. Ciò ci pone di fronte a due problemi:

Espressioni aritmetiche e relazionali: problemi

L'introduzione del sottotipo [`nat`] di [`int`] ci obbliga ad apportare delle modifiche alle espressioni binarie di Small21. Ciò ci pone di fronte a due problemi:

1. Quali tipi sono ammessi per i due operandi
2. Quale tipo inferire al risultato dell'operazione

Espressioni aritmetiche e relazionali: problemi

L'introduzione del sottotipo [`nat`] di [`int`] ci obbliga ad apportare delle modifiche alle espressioni binarie di Small21. Ciò ci pone di fronte a due problemi:

1. Quali tipi sono ammessi per i due operandi
2. Quale tipo inferire al risultato dell'operazione

Il problema riguarda la *segnatura* delle operazioni.

Espressioni aritmetiche e relazionali: problemi

L'introduzione del sottotipo [nat] di [int] ci obbliga ad apportare delle modifiche alle espressioni binarie di Small21. Ciò ci pone di fronte a due problemi:

1. Quali tipi sono ammessi per i due operandi
2. Quale tipo inferire al risultato dell'operazione

Il problema riguarda la *segnatura* delle operazioni. In Small21, prima di questa modifica:

$$\mathcal{O}_2 = \{+, -, *, \texttt{div}, ==, >, <, \texttt{or}, \texttt{and}\}$$

$$\Delta_0(\text{op}) = \begin{cases} [\texttt{abs}]\ [\texttt{int}][::][\texttt{int}][::][\texttt{int}] & \text{iff} \quad \text{op} \in \{+, -, *\} \\ [\texttt{abs}]\ [\texttt{bool}][::][\texttt{int}][::][\texttt{int}] & \text{iff} \quad \text{op} \in \{==, >, <\} \end{cases}$$

Espressioni aritmetiche e relazionali: problemi

L'introduzione del sottotipo [nat] di [int] ci obbliga ad apportare delle modifiche alle espressioni binarie di Small21. Ciò ci pone di fronte a due problemi:

1. Quali tipi sono ammessi per i due operandi
2. Quale tipo inferire al risultato dell'operazione

Il problema riguarda la *segnatura* delle operazioni. In Small21, prima di questa modifica:

$$\mathcal{O}_2 = \{+, -, *, \texttt{div}, ==, >, <, \texttt{or}, \texttt{and}\}$$

$$\Delta_0(\text{op}) = \begin{cases} [\texttt{abs}]\ [\texttt{int}][::][\texttt{int}][::][\texttt{int}] & \text{iff} \quad \text{op} \in \{+, -, *\} \\ [\texttt{abs}]\ [\texttt{bool}][::][\texttt{int}][::][\texttt{int}] & \text{iff} \quad \text{op} \in \{==, >, <\} \end{cases}$$

Osservazione

Il secondo problema non riguarda le operazioni relazionali $==, <, >$.

Operazioni aritmetiche: discussione di diversi approcci

Vi sono molti modi diversi di procedere, ciascuno con vantaggi e svantaggi, ciascuno legittimo o meno per diversi motivi. È interessante discuterne alcuni, prima di analizzare la soluzione adottata.

Operazioni aritmetiche: discussione di diversi approcci

1. Ogni op è overloaded, e bisogna attenersi rigidamente alla segnatura: sono semanticamente errate operazioni che coinvolgono operandi di tipi diversi.

Vantaggi: di chiara semantica, rispetta la natura type safe di Small21.
Svantaggi: non rispetta la compatibilità tra [`nat`] e [`int`].

Operazioni aritmetiche: discussione di diversi approcci

1. Ogni op è overloaded, e bisogna attenersi rigidamente alla segnatura: sono semanticamente errate operazioni che coinvolgono operandi di tipi diversi.
2. op (aritmetica) assegna a qualsiasi risultato il tipo `int`, ignorando la relazione tipo/sottotipo

Vantaggi: di chiara semantica, nessun overloading, non produce errori di tipo
Svantaggi: avremmo introdotto un tipo senza alcuna operazione che ne manipola i valori!

Operazioni aritmetiche: discussione di diversi approcci

1. Ogni op è overloaded, e bisogna attenersi rigidamente alla segnatura: sono semanticamente errate operazioni che coinvolgono operandi di tipi diversi.
2. op (aritmetica) assegna a qualsiasi risultato il tipo `int`, ignorando la relazione tipo/sottotipo
3. Solo + e $*$ sono overloaded

Vantaggi: matematicamente coerente, risponde direttamente alle problematiche della strategia precedente
Svantaggi: semantica poco chiara: due sole operazioni sono overloaded. Un programmatore potrebbe aspettarsi di manipolare (o confrontare) valori di tipo `nat` con più operazioni senza incorrere in frustranti errori di tipo.

Operazioni aritmetiche: discussione di diversi approcci

1. Ogni op è overloaded, e bisogna attenersi rigidamente alla segnatura: sono semanticamente errate operazioni che coinvolgono operandi di tipi diversi.
2. op (aritmetica) assegna a qualsiasi risultato il tipo `int`, ignorando la relazione tipo/sottotipo
3. Solo + e $*$ sono overloaded
4. Ciascuna op è overloaded. Al momento dell'applicazione cerchiamo (dinamicamente e tramite un algoritmo) quella con segnatura "più prossima", eventualmente promuovendo gli operandi seguendo una strategia leftmost-first

Vantaggi: possiamo operare sui `nat` con ogni operazione, strategia usata da gran parte dei linguaggi oo, di chiara semantica. È possibile inferire staticamente il tipo del risultato dell'operazione
Svantaggi: semiformale, inferisce il tipo più "debole" al risultato e questo può generare fastidiosi errori di tipo (risolvibili con un cast)

Operazioni aritmetiche: discussione di diversi approcci

1. Ogni op è overloaded, e bisogna attenersi rigidamente alla segnatura: sono semanticamente errate operazioni che coinvolgono operandi di tipi diversi.
2. op (aritmetica) assegna a qualsiasi risultato il tipo `int`, ignorando la relazione tipo/sottotipo
3. Solo + e $*$ sono overloaded
4. Ciascuna op è overloaded. Al momento dell'applicazione cerchiamo (dinamicamente e tramite un algoritmo) quella con segnatura "più prossima", eventualmente promuovendo gli operandi seguendo una strategia leftmost-first
5. Inferire "il tipo più stretto possibile" al risultato di ciascuna operazione (tutte overloaded)

Vantaggi: possiamo operare sui `nat` con ogni operazione, non produce errori di tipo (se ben implementata). Evita molti fastidiosi errori di tipo al programmatore, evitando di dover fargli fare molti cast espliciti
Svantaggi: Regola troppo vaga, di non chiara semantica. Necessita di essere trattata dinamicamente

Operazioni binarie: soluzione adottata in Small21 (2)

Abbiamo deciso di adottare una strategia simile alla n°5.
Le denotazioni delle operazioni sono adesso *overloaded*: lo stesso nome denota operazioni diverse (per segnatura, ma non solo (cf. oss)).
È mantenuta la possibilità di fare cast esplicito, tramite l'apposita espressione.

Di fronte a operandi di tipo diverso ma compatibile, Small21 cercherà di fare *downcast* sull'operando di tipo più debole.

Operazioni binarie: soluzione adottata in Small21 (2)

Abbiamo deciso di adottare una strategia simile alla n°5.
Le denotazioni delle operazioni sono adesso *overloaded*: lo stesso nome denota operazioni diverse (per segnatura, ma non solo (cf. oss)).
È mantenuta la possibilità di fare cast esplicito, tramite l'apposita espressione.

Di fronte a operandi di tipo diverso ma compatibile, Small21 cercherà di fare *downcast* sull'operando di tipo più debole.

- Se il cast ha successo, applicherà l'operazione con segnatura $[\texttt{abs}]\ \texttt{t[::][nat][::][nat]}$
- Altrimenti, promuoverà (*upcast*) l'altro operando e applicherà l'altra operazione

Operazioni binarie: soluzione adottata in Small21 (2)

Osservazione

In realtà non vi è bisogno di fare *upcast*: tale situazione si può presentare solo se un operando ha tipo [`int`] e l'altro [`nat`], ma [`nat`] $<$ [`int`], e i valori hanno la stessa rappresentazione in memoria.

Operazioni binarie: soluzione adottata in Small21 (2)

Osservazione

In realtà non vi è bisogno di fare *upcast*: tale situazione si può presentare solo se un operando ha tipo [`int`] e l'altro [`nat`], ma [`nat`] $<$ [`int`], e i valori hanno la stessa rappresentazione in memoria.

Osservazione

Va prestata attenzione all'operazione "$-$" di sottrazione. Adesso $-_{\texttt{nat}}$ e $-_{\texttt{int}}$ differiscono per funzione calcolata oltre che per segnatura! Infatti non sempre la differenza di due naturali è un naturale. Questo richiede un controllo aggiuntivo a runtime.

Overloading

Operazioni binarie:

$$\mathcal{O}_2 = \{+, -, *, ==, >, <, \texttt{or}\}$$

$$+ = \{+_{\texttt{nat}}, +_{\texttt{int}}\},\ - = \{-_{\texttt{nat}}, -_{\texttt{int}}\},\ * = \{*_{\texttt{nat}}, *_{\texttt{int}}\}$$

$$== = \{==_{\texttt{nat}}, ==_{\texttt{int}}, ==_{\texttt{bool}}\},\ > = \{>_{\texttt{nat}}, >_{\texttt{int}}\},\ < = \{<_{\texttt{nat}}, <_{\texttt{int}}\}$$

Overloading

Operazioni binarie:

$$\mathcal{O}_2 = \{+, -, *, ==, >, <, \texttt{or}\}$$

$$+ = \{+_{\texttt{nat}}, +_{\texttt{int}}\},\ - = \{-_{\texttt{nat}}, -_{\texttt{int}}\},\ * = \{*_{\texttt{nat}}, *_{\texttt{int}}\}$$

$$== = \{==_{\texttt{nat}}, ==_{\texttt{int}}, ==_{\texttt{bool}}\},\ > = \{>_{\texttt{nat}}, >_{\texttt{int}}\},\ < = \{<_{\texttt{nat}}, <_{\texttt{int}}\}$$

$\mathcal{O}_2 \subseteq \texttt{Ide}$ è adesso un sottoinsieme di *nomi*. La scrittura $\texttt{op} = \{\texttt{op}_{\texttt{nat}}, \texttt{op}_{\texttt{int}}\}$ esprime il fatto che op denota le operazioni $\texttt{op}_{\texttt{nat}}$ e $\texttt{op}_{\texttt{int}}$

Overloading

Operazioni binarie:

$$\mathcal{O}_2 = \{+, -, *, ==, >, <, \texttt{or}\}$$

$$+ = \{+_{\texttt{nat}}, +_{\texttt{int}}\},\ - = \{-_{\texttt{nat}}, -_{\texttt{int}}\},\ * = \{*_{\texttt{nat}}, *_{\texttt{int}}\}$$

$$== = \{==_{\texttt{nat}}, ==_{\texttt{int}}, ==_{\texttt{bool}}\},\ > = \{>_{\texttt{nat}}, >_{\texttt{int}}\},\ < = \{<_{\texttt{nat}}, <_{\texttt{int}}\}$$

Segnature:

$$\texttt{op}_{\texttt{int}} \begin{cases} [\texttt{abs}]\,[\texttt{int}][::][\texttt{int}][::][\texttt{int}] & \text{iff} \quad \texttt{op} \in \{+, -, *\} \\ [\texttt{abs}]\,[\texttt{bool}][::][\texttt{int}][::][\texttt{int}] & \text{iff} \quad \texttt{op} \in \{>, <\} \end{cases}$$

$$\texttt{op}_{\texttt{nat}} \begin{cases} [\texttt{abs}]\,[\texttt{nat}][::][\texttt{nat}][::][\texttt{nat}] & \text{iff} \quad \texttt{op} \in \{+, -, *\} \\ [\texttt{abs}]\,[\texttt{bool}][::][\texttt{nat}][::][\texttt{nat}] & \text{iff} \quad \texttt{op} \in \{>, <\} \end{cases}$$

$$\Delta(==_{\texttt{t}}) = [\texttt{abs}]\,[\texttt{bool}][::]\texttt{t}[::]\texttt{t} \qquad \texttt{t} \in \{[\texttt{int}], [\texttt{nat}], [\texttt{bool}]\}$$

Nuovo sistema di inferenza $\mathtt{OP}_\mathtt{S}$ (1)

Introduciamo un nuovo sistema di inferenza, $\mathtt{OP}_\mathtt{S}$, per trattare la scelta dell'operazione corretta da eseguire in base al tipo degli operandi.

Nuovo sistema di inferenza $\texttt{OP}_\texttt{S}$ (1)

Introduciamo un nuovo sistema di inferenza, $\texttt{OP}_\texttt{S}$, per trattare la scelta dell'operazione corretta da eseguire in base al tipo degli operandi.

Operativamente, ad $\texttt{OP}_\texttt{S}$ non associamo una funzione semantica nell'interprete. Infatti `nat` e `int` condividono la rappresentazione in memoria, e le operazioni fornite dal RTS di OCaml non hanno problemi a manipolare valori di tali tipi.

Per chiarezza espositiva,

$$\texttt{OP}_\texttt{S} : \mathcal{O}_2 \to \texttt{Type} \to \texttt{State} \longrightarrow \texttt{Type} * \texttt{Operators} * \texttt{State}$$

dove `Operators` è la famiglia degli operatori (binari) denotati dagli elementi di $\mathcal{O}_2$.

Nuovo sistema di inferenza $\mathtt{OP_S}$ (2)

La scrittura $\langle \mathtt{op} \prec \mathtt{t}, \sigma \rangle \rightarrow_{\mathtt{OP_S}} \lfloor \mathtt{t}_o, \mathtt{op}', \sigma \rfloor$ vuol dire che, nel contesto dato dallo stato σ, in base al tipo $\mathtt{t}$ l'operazione che il nome $\mathtt{op}$ sta denotando è $\mathtt{op}'$, la quale ha tipo $\mathtt{t}_o$ (dato dalla segnatura).

Nuovo sistema di inferenza $\mathtt{OP_S}$ (2)

La scrittura $\langle \mathtt{op} \prec \mathtt{t}, \sigma \rangle \rightarrow_{\mathtt{OP_S}} \lfloor \mathtt{t}_o, \mathtt{op}', \sigma \rfloor$ vuol dire che, nel contesto dato dallo stato σ, in base al tipo $\mathtt{t}$ l'operazione che il nome $\mathtt{op}$ sta denotando è $\mathtt{op}'$, la quale ha tipo $\mathtt{t}_o$ (dato dalla segnatura).

Osservazione

Lo stato non viene modificato, ha il solo scopo di fornire un contesto nel quale l'operazione è invocata. Per determinare la segnatura adatta al contesto dell'invocazione dell'operazione (è presente nell'ambiente standard, fornito dall'RTS di AM21) è sufficiente il tipo di un operando (ipotesi: operandi di tipo omogeneo).

Nuovo sistema di inferenza $\mathtt{OP_S}$ (3)

Sistema Y:

$$[\text{Y99}]\ \frac{\mathtt{op} \in \mathcal{O}_2 \qquad \mathtt{t_1} \in \mathtt{Simple} \qquad \mathtt{op} = \{\mathtt{op_1}, \ldots, \mathtt{op_n}\} \quad \Delta(\mathtt{op_i}) = \big[\mathtt{abs}\big]\ \mathtt{t}[::]\mathtt{t_1'}[::]\mathtt{t_1'} \qquad \mathtt{i} \in \{1, \ldots, n\} \qquad \mathtt{t_1} = \mathtt{t_1'} \qquad \big[\mathtt{abs}\big]\ \mathtt{t}[::]\mathtt{t_1'}[::]\mathtt{t_1'} = \mathtt{t}_o}{\langle \mathtt{op} \prec \mathtt{t_1}, \mathtt{Y}_\rho \rangle \rightarrow_{\mathtt{Y}} (\mathtt{t}_o, \mathtt{Y}_\rho)}$$

Errori di tipo:

$$[\text{E900}]\ \frac{\mathtt{op} \notin \mathcal{O}_2}{\langle \mathtt{op} \prec \mathtt{t_1}, \mathtt{Y}_\rho \rangle \rightarrow_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)} \qquad [\text{E901}]\ \frac{\mathtt{t_1} \notin \mathtt{Simple}}{\langle \mathtt{op} \prec \mathtt{t_1}, \mathtt{Y}_\rho \rangle \rightarrow_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

$$\ldots$$

$$[\text{E902}]\ \frac{\Delta(\mathtt{op_i}) = \big[\mathtt{abs}\big]\ \mathtt{t}[::]\mathtt{t_1'}[::]\mathtt{t_1'} \quad \mathtt{t_1} \neq \mathtt{t_1'} \quad \forall \mathtt{i} \in \{1, \ldots, n\}}{\langle \mathtt{op} \prec \mathtt{t_1}, \mathtt{Y}_\rho \rangle \rightarrow_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

Nuovo sistema di inferenza $\mathtt{OP_S}$ (3)

Regole di Inferenza:

$$[\text{Z1}]\frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \qquad \mathtt{t_1} \in \mathtt{Simple} \\ \mathtt{op} = \{\mathtt{op_1}, \ldots, \mathtt{op_n}\} \quad \Delta(\mathtt{op_i}) = \left[\mathtt{abs}\right] \mathtt{t}[::]\mathtt{t}'_1[::]\mathtt{t}'_1 \\ \mathtt{i} \in \{1, \ldots, n\} \qquad \mathtt{t_1} = \mathtt{t}'_1 \qquad \left[\mathtt{abs}\right] \mathtt{t}[::]\mathtt{t}'_1[::]\mathtt{t}'_1 = \mathtt{t}_o\end{array}}{\langle \mathtt{op} \prec \mathtt{t_1}, \sigma\rangle \rightarrow_{\mathtt{OP_S}} \lfloor \mathtt{t}_o, \mathtt{op_i}, \sigma \rfloor}$$

Assunzioni: le operazioni ammettono solo operandi di tipo omogeneo. Abbiamo fatto uso di forme proposizionali per praticità di notazione.

Operazioni: modifica regole per EXP - Sistema Y (1)

Modifichiamo la regola [Y17]

Sistema dei tipi Y:

$$[\text{Y17.1'}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_1, \mathtt{Y}_\rho) \\ \langle \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_2, \mathtt{Y}_\rho) \\ \mathtt{t}_1 \leq \mathtt{t}_2 \qquad [\mathtt{cast}]\ \mathtt{t}_1\ \mathtt{e}_2 = \mathtt{e}_2' \\ \langle \mathtt{e}_2', \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{e}, \mathtt{Y}_\rho) \qquad \mathtt{t}_\mathtt{e} = \mathtt{t}_1 \\ \langle \mathtt{op} \prec \mathtt{t}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_0, \mathtt{Y}_\rho) \\ \mathtt{t}_o = [\mathtt{abs}]\ \mathtt{t}[::]\mathtt{t}_1'[::]\mathtt{t}_1' \\ \mathtt{op} \neq - \vee \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)} \quad (\mathtt{op}' = -_{\mathtt{nat}}\ \mathtt{and}\ \mathtt{DynFeasCheck})$$

$$[\text{Y17.1''}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_1, \mathtt{Y}_\rho) \\ \langle \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_2, \mathtt{Y}_\rho) \\ \mathtt{t}_2 \leq \mathtt{t}_1 \qquad [\mathtt{cast}]\ \mathtt{t}_2\ \mathtt{e}_1 = \mathtt{e}_1' \\ \langle \mathtt{e}_1', \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{e}, \mathtt{Y}_\rho) \qquad \mathtt{t}_\mathtt{e} = \mathtt{t}_2 \\ \langle \mathtt{op} \prec \mathtt{t}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_0, \mathtt{Y}_\rho) \\ \mathtt{t}_o = [\mathtt{abs}]\ \mathtt{t}[::]\mathtt{t}_2'[::]\mathtt{t}_2' \\ \mathtt{op} \neq - \vee \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)} \quad (\mathtt{op}' = -_{\mathtt{nat}}\ \mathtt{and}\ \mathtt{DynFeasCheck})$$

`DynFeasCheck`: controllo a runtime di ammissibilità dell'operazione $-$ tra numeri naturali.

Operazioni: modifica regole per EXP - Sistema Y (1)

Modifichiamo la regola [Y17]

Sistema dei tipi Y:

$$[\text{Y17.2'}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_1, \mathtt{Y}_\rho) \\ \langle \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_2, \mathtt{Y}_\rho) \\ \mathtt{t}_1 \leq \mathtt{t}_2 \qquad [\mathtt{cast}]\ \mathtt{t}_1\ \mathtt{e}_2 = \mathtt{e}_2' \\ \langle \mathtt{e}_2', \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{e}, \mathtt{Y}_\rho) \\ \mathtt{t}_\mathtt{e} = [\mathtt{terr}] \\ \langle \mathtt{op} \prec \mathtt{t}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_0, \mathtt{Y}_\rho) \\ \mathtt{t}_o = \big[\mathtt{abs}\big]\ \mathtt{t}[::]\mathtt{t}_2'[::]\mathtt{t}_2' \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)}$$

$$[\text{Y17.2''}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_1, \mathtt{Y}_\rho) \\ \langle \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_2, \mathtt{Y}_\rho) \\ \mathtt{t}_2 \leq \mathtt{t}_1 \qquad [\mathtt{cast}]\ \mathtt{t}_2\ \mathtt{e}_1 = \mathtt{e}_1' \\ \langle \mathtt{e}_1', \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{e}, \mathtt{Y}_\rho) \\ \mathtt{t}_\mathtt{e} = [\mathtt{terr}] \\ \langle \mathtt{op} \prec \mathtt{t}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_0, \mathtt{Y}_\rho) \\ \mathtt{t}_o = \big[\mathtt{abs}\big]\ \mathtt{t}[::]\mathtt{t}_1'[::]\mathtt{t}_1' \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)}$$

Operazioni: modifica regole per EXP - Sistema Y (1)

$$[\text{Y17.2'}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_1, \mathtt{Y}_\rho) \\ \langle \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_2, \mathtt{Y}_\rho) \\ \mathtt{t}_1 \leq \mathtt{t}_2 \qquad [\mathtt{cast}]\ \mathtt{t}_1\ \mathtt{e}_2 = \mathtt{e}_2' \\ \langle \mathtt{e}_2', \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{e}, \mathtt{Y}_\rho) \\ \mathtt{t}_\mathtt{e} = [\mathtt{terr}] \\ \langle \mathtt{op} \prec \mathtt{t}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_0, \mathtt{Y}_\rho) \\ \mathtt{t}_o = [\mathtt{abs}]\ \mathtt{t}[::]\mathtt{t}_2'[::]\mathtt{t}_2' \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)}$$

$$[\text{Y17.2''}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_1, \mathtt{Y}_\rho) \\ \langle \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_2, \mathtt{Y}_\rho) \\ \mathtt{t}_2 \leq \mathtt{t}_1 \qquad [\mathtt{cast}]\ \mathtt{t}_2\ \mathtt{e}_1 = \mathtt{e}_1' \\ \langle \mathtt{e}_1', \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{e}, \mathtt{Y}_\rho) \\ \mathtt{t}_\mathtt{e} = [\mathtt{terr}] \\ \langle \mathtt{op} \prec \mathtt{t}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_0, \mathtt{Y}_\rho) \\ \mathtt{t}_o = [\mathtt{abs}]\ \mathtt{t}[::]\mathtt{t}_1'[::]\mathtt{t}_1' \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)}$$

Osservazione

$\mathtt{Cast}(\mathtt{t}_1, \mathtt{exp})$ restituisce errore se $\mathtt{t}_1 = [\mathtt{bool}]$ o se (il valore di) exp non è compatibile con tipo [nat]. Nel primo caso, allora $\mathtt{t}_2 = [\mathtt{bool}]$, nel secondo bisognerebbe fare upcast all' operando con tipo più debole, $\mathtt{t}_2$.

Operazioni: modifica regole per EXP - Sistema Y (1)

$$[\text{Y17.2'}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_1, \mathtt{Y}_\rho) \\ \langle \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_2, \mathtt{Y}_\rho) \\ \mathtt{t}_1 \leq \mathtt{t}_2 \qquad [\mathtt{cast}]\ \mathtt{t}_1\ \mathtt{e}_2 = \mathtt{e}_2' \\ \langle \mathtt{e}_2', \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{e}, \mathtt{Y}_\rho) \\ \mathtt{t}_\mathtt{e} = [\mathtt{terr}] \\ \langle \mathtt{op} \prec \mathtt{t}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_0, \mathtt{Y}_\rho) \\ \mathtt{t}_o = \left[\mathtt{abs}\right] \mathtt{t}[::]\mathtt{t}_2'[::]\mathtt{t}_2' \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)}$$

$$[\text{Y17.2''}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_1, \mathtt{Y}_\rho) \\ \langle \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_2, \mathtt{Y}_\rho) \\ \mathtt{t}_2 \leq \mathtt{t}_1 \qquad [\mathtt{cast}]\ \mathtt{t}_2\ \mathtt{e}_1 = \mathtt{e}_1' \\ \langle \mathtt{e}_1', \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_\mathtt{e}, \mathtt{Y}_\rho) \\ \mathtt{t}_\mathtt{e} = [\mathtt{terr}] \\ \langle \mathtt{op} \prec \mathtt{t}_1, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}_0, \mathtt{Y}_\rho) \\ \mathtt{t}_o = \left[\mathtt{abs}\right] \mathtt{t}[::]\mathtt{t}_1'[::]\mathtt{t}_1' \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \mathtt{Y}_\rho \rangle \to_{\mathtt{Y}} (\mathtt{t}, \mathtt{Y}_\rho)}$$

Osservazione

In [Y17.2'] e [Y17.2''] non c'è bisogno del controllo (dinamico) di ammissibilità dell'operazione $-_{\mathtt{nat}}$ perché l'operazione $-$ è sempre ben definita sugli interi (che è il tipo necessariamente inferito se le premesse sono verificate)

Operazioni: modifica regole per EXP - Sistema Y (2)

Errori di Tipo:

$$\cdots$$

$$[\text{E25'}]\ \frac{\mathtt{t}_1 \not\preceq \mathtt{t}_2 \text{ and } \mathtt{t}_2 \not\preceq \mathtt{t}_1}{\langle \mathtt{e}_1\,[\mathtt{op}]\,\mathtt{e}_2, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

$$\cdots$$

$$[\text{E26'}]\ \frac{\langle \mathtt{op} \prec \mathtt{t}_1, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}{\langle \mathtt{e}_1\,[\mathtt{op}]\,\mathtt{e}_2, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

$$\cdots$$

$$[\text{E26''}]\ \frac{\langle \mathtt{op} \prec \mathtt{t}_2, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}{\langle \mathtt{e}_1\,[\mathtt{op}]\,\mathtt{e}_2, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

$$\cdots$$

$$[\text{E27}]\ \frac{\mathtt{op}' = -_{\mathtt{nat}} \text{ and } \mathtt{DynamicFeasibilityCheck:Failure}}{\langle \mathtt{e}_1\,[\mathtt{op}]\,\mathtt{e}_2, \mathtt{Y}_\rho\rangle \to_{\mathtt{Y}} ([\mathtt{terr}], \mathtt{Y}_\rho)}$$

Operazioni: modifica regole per EXP - Sistema $\mathtt{SEM}_{\mathtt{EXP}}$

Modifichiamo la regola [X9]

Regole di inferenza:

$$[\text{X9.1'}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \sigma \rangle \to \lfloor \mathtt{t}_1,\ \mathtt{v}_1,\ \sigma_1 \rfloor \\ \langle \mathtt{e}_2, \sigma_1 \rangle \to \lfloor \mathtt{t}_2,\ \mathtt{v}_2,\ \sigma_2 \rfloor \\ \mathtt{t}_1 \leq \mathtt{t}_2 \qquad [\mathtt{cast}]\ \mathtt{t}_1\ \mathtt{e}_2 = \mathtt{e}_2' \\ \langle \mathtt{e}_2', \sigma_2 \rangle \to_{\mathtt{SEM}_{\mathtt{EXP}}} \lfloor \mathtt{t}_e,\ \mathtt{v}_2',\ \sigma_2' \rfloor \\ \mathtt{t}_e = \mathtt{t}_1 \\ \langle \mathtt{op} \prec \mathtt{t}_1, \sigma_2' \rangle \to_{\mathtt{OP_S}} \lfloor \mathtt{t}_o,\ \mathtt{op}',\ \sigma_3 \rfloor \\ \mathtt{t}_o = [\mathtt{abs}]\ \mathtt{t}[::]\mathtt{t}_1'[::]\mathtt{t}_1' \\ \overline{\mathtt{op}'}(\mathtt{v}_1, \mathtt{v}_2') = \mathtt{v} \\ \mathtt{op} \neq - \vee \\ (\mathtt{op}' = -_{\mathtt{nat}}\ \mathtt{and}\ \mathtt{v} \geq 0)\end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \sigma \rangle \to \lfloor \mathtt{t},\ \mathtt{v},\ \sigma_3 \rfloor}$$

$$[\text{X9.1''}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \sigma \rangle \to \lfloor \mathtt{t}_1,\ \mathtt{v}_1,\ \sigma_1 \rfloor \\ \langle \mathtt{e}_2, \sigma_1 \rangle \to \lfloor \mathtt{t}_2,\ \mathtt{v}_2,\ \sigma_2 \rfloor \\ \mathtt{t}_2 \leq \mathtt{t}_1 \qquad [\mathtt{cast}]\ \mathtt{t}_2\ \mathtt{e}_1 = \mathtt{e}_1' \\ \langle \mathtt{e}_1', \sigma_2 \rangle \to_{\mathtt{SEM}_{\mathtt{EXP}}} \lfloor \mathtt{t}_e,\ \mathtt{v}_1',\ \sigma_2' \rfloor \\ \mathtt{t}_e = \mathtt{t}_2 \\ \langle \mathtt{op} \prec \mathtt{t}_2, \sigma_2' \rangle \to_{\mathtt{OP_S}} \lfloor \mathtt{t}_o,\ \mathtt{op}',\ \sigma_3 \rfloor \\ \mathtt{t}_o = [\mathtt{abs}]\ \mathtt{t}[::]\mathtt{t}_2'[::]\mathtt{t}_2' \\ \overline{\mathtt{op}'}(\mathtt{v}_1', \mathtt{v}_2) = \mathtt{v} \\ \mathtt{op} \neq - \vee \\ (\mathtt{op}' = -_{\mathtt{nat}}\ \mathtt{and}\ \mathtt{v} \geq 0)\end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \sigma \rangle \to \lfloor \mathtt{t},\ \mathtt{v},\ \sigma_3 \rfloor}$$

Operazioni: modifica regole per EXP - Sistema $\mathtt{SEM}_{\mathtt{EXP}}$

Modifichiamo la regola [X9]

Regole di inferenza:

$$[\text{X9.2'}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \sigma \rangle \to \lfloor \mathtt{t}_1,\ \mathtt{v}_1,\ \sigma_1 \rfloor \\ \langle \mathtt{e}_2, \sigma_1 \rangle \to \lfloor \mathtt{t}_2,\ \mathtt{v}_2,\ \sigma_2 \rfloor \\ \mathtt{t}_1 \leq \mathtt{t}_2 \qquad [\mathtt{cast}]\ \mathtt{t}_1\ \mathtt{e}_2 = \mathtt{e}_2' \\ \langle \mathtt{e}_2', \sigma_2 \rangle \to_{\mathtt{SEM}_{\mathtt{EXP}}} \lfloor \mathtt{t}_e,\ \mathtt{v}_2',\ \sigma_2' \rfloor \\ \mathtt{t}_e = [\mathtt{terr}] \\ \langle \mathtt{op} \prec \mathtt{t}_2, \sigma_2' \rangle \to_{\mathtt{OP_S}} \lfloor \mathtt{t}_o,\ \mathtt{op}',\ \sigma_3 \rfloor \\ \mathtt{t}_o = [\mathtt{abs}]\ \mathtt{t}[::]\mathtt{t}_2'[::]\mathtt{t}_2' \\ \langle [\mathtt{cast}]\ \mathtt{t}_2\ \mathtt{e}_1, \sigma_3 \rangle \to_{\mathtt{SEM}_{\mathtt{EXP}}} \lfloor \mathtt{t}_2,\ \mathtt{v}_1',\ \sigma_3' \rfloor \\ \overline{\mathtt{op}'}(\mathtt{v}_1', \mathtt{v}_2) = \mathtt{v} \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \sigma \rangle \to \lfloor \mathtt{t},\ \mathtt{v},\ \sigma_3' \rfloor}$$

Operazioni: modifica regole per EXP - Sistema $\mathtt{SEM}_{\mathtt{EXP}}$

Modifichiamo la regola [X9]

Regole di inferenza:

$$[\text{X9.2''}]\ \frac{\begin{array}{c}\mathtt{op} \in \mathcal{O}_2 \\ \langle \mathtt{e}_1, \sigma \rangle \to \lfloor \mathtt{t}_1, \mathtt{v}_1, \sigma_1 \rfloor \\ \langle \mathtt{e}_2, \sigma_1 \rangle \to \lfloor \mathtt{t}_2, \mathtt{v}_2, \sigma_2 \rfloor \\ \mathtt{t}_2 \leq \mathtt{t}_1 \qquad [\mathtt{cast}]\ \mathtt{t}_2\ \mathtt{e}_1 = \mathtt{e}_1' \\ \langle \mathtt{e}_1', \sigma_2 \rangle \to_{\mathtt{SEM}_{\mathtt{EXP}}} \lfloor \mathtt{t}_e, \mathtt{v}_1', \sigma_2' \rfloor \\ \mathtt{t}_e = [\mathtt{terr}] \\ \langle \mathtt{op} \prec \mathtt{t}_1, \sigma_2' \rangle \to_{\mathtt{OP_S}} \lfloor \mathtt{t}_o, \mathtt{op}', \sigma_3 \rfloor \\ \mathtt{t}_o = [\mathtt{abs}]\ \mathtt{t}[::]\mathtt{t}_1'[::]\mathtt{t}_1' \\ \langle [\mathtt{cast}]\ \mathtt{t}_1\ \mathtt{e}_2, \sigma_3 \rangle \to_{\mathtt{SEM}_{\mathtt{EXP}}} \lfloor \mathtt{t}_1, \mathtt{v}_2', \sigma_3' \rfloor \\ \overline{\mathtt{op}'}(\mathtt{v}_1, \mathtt{v}_2') = \mathtt{v} \end{array}}{\langle \mathtt{e}_1\ [\mathtt{op}]\ \mathtt{e}_2, \sigma \rangle \to \lfloor \mathtt{t}, \mathtt{v}, \sigma_3' \rfloor}$$

Modifiche all'interprete (1)

```
| Times(e1,e2) ->
    (let (t1,v1,sg1) = expSem e1 (sk,mu) in
     let (t2,v2,sg2) = expSem e2 sg1 in
     match (t1,t2,v1,v2) with
        | (Int,Int,Ival n1,Ival n2)
            -> (Int,Ival(n1*n2),sg2)
        | (Nat,Nat, Ival n1, Ival n2)
            -> (Nat, Ival (n1*n2), sg2)
        | (Int, Nat, Ival n1, Ival n2)
            ->  try
                 let e1p = Cast(t2,e1) in
                 let (te,v1p,sg2p) = expSem e1p sg2 in
                 let Ival n1p = v1p in
                 (te,Ival(n1p*n2),sg2p)
                with
                | TypeErrorE("E802: expSem: -",_)
                    -> let e2p = Cast(t1,e2) in
                       let (_,v2p,sg3p) = expSem e2p sg2 in
                       let Ival n2p = v2p in
                       (t1,Ival(n1*n2p),sg3p)
        | (Nat,Int,Ival n1, Ival n2)
            ->  try
                 let e2p = Cast(t1,e2) in
                 let (te,v2p,sg2p) = expSem e2p sg2 in
                 let Ival n2p = v2p in
                 (te,Ival(n1*n2p),sg2p)
                with
                | TypeErrorE("E802: expSem: -",_)
                    -> let e1p = Cast(t2,e1) in
                       let (_,v1p,sg3p) = expSem e1p sg2 in
                       let Ival n1p = v1p in
                       (t2,Ival(n1p*n2),sg3p)
        | (_,Int,_,Ival n2)         (* Still to fix *)
            -> raise(TypeErrorE("E25': expSem - ",exp))
        | _ -> raise(TypeErrorE("E26: expSem - ",exp)))
```

Figura: *Espressione * nell'interprete Small21*

```
| Minus(e1,e2) ->
    (let (t1,v1,sg1) = expSem e1 (sk,mu) in
     let (t2,v2,sg2) = expSem e2 sg1 in
     match (t1,t2,v1,v2) with
        | (Int,Int,Ival n1,Ival n2)
            -> (Int,Ival(n1-n2),sg2)
        | (Nat,Nat, Ival n1, Ival n2) when (n1 >= n2)
            -> (Nat, Ival (n1-n2), sg2)
        | (Nat,Nat,_,_)
            -> raise(TypeErrorE("E27: expSem - ",exp))
        | (Int, Nat, Ival n1, Ival n2)
            -> (try
                    let e1p = Cast(t2,e1) in
                    let (te,v1p,sg2p) = expSem e1p sg2 in
                    let Ival n1p = v1p in
                    if (n1p >= n2) then (te,Ival(n1p-n2),sg2p)
                    else raise(TypeErrorE("E27: expSem - ",exp))
                with
                | TypeErrorE("E802: expSem: -",_)
                    -> (let e2p = Cast(t1,e2) in
                        let (_,v2p,sg3p) = expSem e2p sg2 in
                        let Ival n2p = v2p in
                        (t1,Ival(n1-n2p),sg3p)))
        | (Nat,Int,Ival n1, Ival n2)
            -> (try
                    let e2p = Cast(t1,e2) in
                    let (te,v2p,sg2p) = expSem e2p sg2 in
                    let Ival n2p = v2p in
                    if (n1 >= n2p) then (te,Ival(n1-n2p),sg2p)
                    else raise(TypeErrorE("E27: expSem - ",exp))
                with
                | TypeErrorE("E802: expSem: -",_)
                    -> (let e1p = Cast(t2,e1) in
                        let (_,v1p,sg3p) = expSem e1p sg2 in
                        let Ival n1p = v1p in
                        (t2,Ival(n1p-n2),sg3p)))
        | (_,Int,_,Ival n2)         (* Still to fix *)
            -> raise(TypeErrorE("E25': expSem - ",exp))
        | _ -> raise(TypeErrorE("E26: expSem - ",exp)))
```

Figura: *Espressione - nell'interprete Small21*

Modifiche all'interprete (2)

```
| Eq(e1,e2) ->  (* changed by me *)
    (let (t1,v1,sg1) = expSem e1 (sk,mu) in
     let (t2,v2,sg2) = expSem e2 sg1 in
     match (t1,t2,v1,v2) with
        | _ when (ysame t1 t2)
          -> (Bool, comp (=) v1 v2,sg2)
        | _ when (ycompatible t1 t2)
          -> (try
                let e2p = Cast(t1,e2) in
                let (_,v2p,sg2p) = expSem e2p sg2 in
                (Bool, comp (=) v1 v2p, sg2p)
              with
              | TypeErrorE("E802: expSem: -",_)
                -> (let e1p = Cast(t2,e1) in
                    let (_,v1p,sg3p) = expSem e1p sg2 in
                    (Bool, comp (=) v1p v2, sg3p)))
        | _ when (ycompatible t2 t1)
          -> (try
                let e1p = Cast(t2,e1) in
                let (_,v1p,sg2p) = expSem e1p sg2 in
                (Bool, comp (=) v1p v2, sg2p)
              with
              | TypeErrorE("E802: expSem: -",_)
                -> (let e2p = Cast(t1,e2) in
                    let (_,v2p,sg3p) = expSem e2p sg2 in
                    (Bool, comp (=) v1 v2p, sg3p)))
        | _ -> raise(TypeErrorE("E25': expSem - ",exp)))
```

Figura: *Espressione == nell'interprete Small21*

```
| LT(e1,e2) ->          (* changed by me *)
    (let (t1,v1,sg1) = expSem e1 (sk,mu) in
     let (t2,v2,sg2) = expSem e2 sg1 in
     match (t1,t2,v1,v2) with
        | (Int,Int,Ival n1,Ival n2)
          -> (Bool,comp (<) n1 n2,sg2)
        | (Nat,Nat, Ival n1, Ival n2)
          -> (Bool,comp (<) n1 n2,sg2)
        | (Int, Nat, Ival n1, Ival n2)
          -> (try
                let e1p = Cast(t2,e1) in
                let (te,v1p,sg2p) = expSem e1p sg2 in
                let Ival n1p = v1p in
                (Bool,comp (<) n1p n2,sg2p)
              with
              | TypeErrorE("E802: expSem: -",_)
                -> (let e2p = Cast(t1,e2) in
                    let (_,v2p,sg3p) = expSem e2p sg2 in
                    let Ival n2p = v2p in
                    (Bool,comp (<) n1 n2p,sg3p)))
        | (Nat,Int,Ival n1, Ival n2)
          -> (try
                let e2p = Cast(t1,e2) in
                let (te,v2p,sg2p) = expSem e2p sg2 in
                let Ival n2p = v2p in
                (Bool,comp (<) n1 n2p,sg2p)
              with
              | TypeErrorE("E802: expSem: -",_)
                -> (let e1p = Cast(t2,e1) in
                    let (_,v1p,sg3p) = expSem e1p sg2 in
                    let Ival n1p = v1p in
                    (Bool,comp (<) n1p n2,sg3p)))
        | (_,Int,_,Ival n2)          (* Still to fix *)
          -> raise(TypeErrorE("E25': expSem - ",exp))
        | _ -> raise(TypeErrorE("E26: expSem - ",exp)))
```

Figura: *Espressione < nell'interprete Small21*

Modifiche all'interprete (2)

```
| Eq(e1,e2) ->  (* changed by me *)
    (let (t1,v1,sg1) = expSem e1 (sk,mu) in
     let (t2,v2,sg2) = expSem e2 sg1 in
     match (t1,t2,v1,v2) with
        | _ when (ysame t1 t2)
            -> (Bool, comp (=) v1 v2,sg2)
        | _ when (ycompatible t1 t2)
            -> (try
                    let e2p = Cast(t1,e2) in
                    let (_,v2p,sg2p) = expSem e2p sg2 in
                    (Bool, comp (=) v1 v2p, sg2p)
                with
                | TypeErrorE("E802: expSem: -",_)
                    -> (let e1p = Cast(t2,e1) in
                        let (_,v1p,sg3p) = expSem e1p sg2 in
                        (Bool, comp (=) v1p v2, sg3p)))
        | _ when (ycompatible t2 t1)
            -> (try
                    let e1p = Cast(t2,e1) in
                    let (_,v1p,sg2p) = expSem e1p sg2 in
                    (Bool, comp (=) v1p v2, sg2p)
                with
                | TypeErrorE("E802: expSem: -",_)
                    -> (let e2p = Cast(t1,e2) in
                        let (_,v2p,sg3p) = expSem e2p sg2 in
                        (Bool, comp (=) v1 v2p, sg3p)))
        | _ -> raise(TypeErrorE("E25': expSem - ",exp)))
```

Figura: *Espressione == nell'interprete Small21*

```
| LT(e1,e2) ->          (* changed by me *)
    (let (t1,v1,sg1) = expSem e1 (sk,mu) in
     let (t2,v2,sg2) = expSem e2 sg1 in
     match (t1,t2,v1,v2) with
        | (Int,Int,Ival n1,Ival n2)
            -> (Bool,comp (<) n1 n2,sg2)
        | (Nat,Nat, Ival n1, Ival n2)
            -> (Bool,comp (<) n1 n2,sg2)
        | (Int, Nat, Ival n1, Ival n2)
            -> (try
                    let e1p = Cast(t2,e1) in
                    let (te,v1p,sg2p) = expSem e1p sg2 in
                    let Ival n1p = v1p in
                    (Bool,comp (<) n1p n2,sg2p)
                with
                | TypeErrorE("E802: expSem: -",_)
                    -> (let e2p = Cast(t1,e2) in
                        let (_,v2p,sg3p) = expSem e2p sg2 in
                        let Ival n2p = v2p in
                        (Bool,comp (<) n1 n2p,sg3p)))
        | (Nat,Int,Ival n1, Ival n2)
            -> (try
                    let e2p = Cast(t1,e2) in
                    let (te,v2p,sg2p) = expSem e2p sg2 in
                    let Ival n2p = v2p in
                    (Bool,comp (<) n1 n2p,sg2p)
                with
                | TypeErrorE("E802: expSem: -",_)
                    -> (let e1p = Cast(t2,e1) in
                        let (_,v1p,sg3p) = expSem e1p sg2 in
                        let Ival n1p = v1p in
                        (Bool,comp (<) n1p n2,sg3p)))
        | (_,Int,_,Ival n2)          (* Still to fix *)
            -> raise(TypeErrorE("E25': expSem - ",exp))
        | _ -> raise(TypeErrorE("E26: expSem - ",exp)))
```

Figura: *Espressione < nell'interprete Small21*

Osservazione

Il sistema di inferenza $\mathtt{OP}_S$ non corrisponde ad una funzione semantica nell'interprete OCaml, le sue operazioni binarie corrispondenti sono compatibili in entrambi i versi, dato che `int` e `nat` condividono la rappresentazione in memoria.

Sommario

Fattoriale - Esempi (1)

Programma che calcola un'applicazione della funzione fattoriale di un *naturale* all'intero 5.

Fattoriale - Esempi (1)

Programma che calcola un'applicazione della funzione fattoriale di un *naturale* all'intero 5.

```
let d1 = Var(Int, "n", N(5)) in
let fp = FP(Value, Nat, "x") in
let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
let espr = Times(Val "x", espaux) in
let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
let bl = BlockP(ED,SeqS(aux1,aux2)) in
let d2 = Pcd(Nat, "fattRic", fp, bl) in
let cas = Val "n" in
let d3 = Var(Int, "nfatt", Apply("fattRic", AP(cas))) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("5fattoriale_ricors_nat", Block(d,UnL ES)) in
printProg prog
```

⇓ `printProg`

```
Program 5fattoriale_ricors_nat{
    int n = 5;
    nat fattRic(value nat x){
          if (x == 0) return 1;
          if ((x == 0) == false) return (x * fattRic((x - 1)));

          }
    int nfatt = fattRic(n);
    }

- : unit/2 = ()
#
```

`progSem` ⟹

```
#     let d1 = Var(Int, "n", N(5)) in
      let fp = FP(Value, Nat, "x") in
      let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
      let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
      let espr = Times(Val "x", espaux) in
      let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
      let bl = BlockP(ED,SeqS(aux1,aux2)) in
      let d2 = Pcd(Nat, "fattRic", fp, bl) in
      let cas = Val "n" in
      let d3 = Var(Int, "nfatt", Apply("fattRic", AP(cas))) in
      let d = SeqD(SeqD(d1,d2),d3) in
      let prog = Prog("5fattoriale_ricors_nat", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorT "E61.1: Trasmission: Type not expected".
#
```

Fattoriale - Esempi (1)

Programma che calcola un'applicazione della funzione fattoriale di un *naturale* all'intero 5.

```
let d1 = Var(Int, "n", N(5)) in
let fp = FP(Value, Nat, "x") in
let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
let espr = Times(Val "x", espaux) in
let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
let bl = BlockP(ED,SeqS(aux1,aux2)) in
let d2 = Pcd(Nat, "fattRic", fp, bl) in
let cas = Val "n" in
let d3 = Var(Int, "nfatt", Apply("fattRic", AP(cas))) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("5fattoriale_ricors_nat", Block(d,UnL ES)) in
printProg prog
```

$\Downarrow$ `printProg`

```
Program 5fattoriale_ricors_nat{
    int n = 5;
    nat fattRic(value nat x){
          if (x == 0) return 1;
          if ((x == 0) == false) return (x * fattRic((x - 1)));

          }
    int nfatt = fattRic(n);
    }

- : unit/2 = ()
#
```

$\xRightarrow{\texttt{progSem}}$

```
#     let d1 = Var(Int, "n", N(5)) in
      let fp = FP(Value, Nat, "x") in
      let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
      let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
      let espr = Times(Val "x", espaux) in
      let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
      let bl = BlockP(ED,SeqS(aux1,aux2)) in
      let d2 = Pcd(Nat, "fattRic", fp, bl) in
      let cas = Val "n" in
      let d3 = Var(Int, "nfatt", Apply("fattRic", AP(cas))) in
      let d = SeqD(SeqD(d1,d2),d3) in
      let prog = Prog("5fattoriale_ricors_nat", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorT "E61.1: Trasmission: Type not expected".
#
```

Come ci aspettavamo, il programma restituisce un errore di trasmissione dei parametri ($[\texttt{int}] \not\leq [\texttt{nat}]$)

Fattoriale - Esempi (2)

Programma che calcola un'applicazione della funzione fattoriale di un *naturale* all'intero -5.

Fattoriale - Esempi (2)

Programma che calcola un'applicazione della funzione fattoriale di un *naturale* all'intero -5.

```
let fp = FP(Value, Nat, "x") in
let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
let espr = Times(Val "x", espaux) in
let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
let bl = BlockP(ED,SeqS(aux1,aux2)) in
let d1 = Pcd(Nat, "fattRic", fp, bl) in
let d2 = Var(Int, "x", Apply("fattRic", AP(N (-5)))) in
let d = SeqD(d1,d2) in
let prog = Prog("-5fatt_ricors_nat", Block(d,UnL ES)) in
printProg prog
```

$\Downarrow$ `printProg`

```
Program -5fatt_ricors_nat{
    nat fattRic(value nat x){
            if (x == 0) return 1;
            if ((x == 0) == false) return (x * fattRic((x - 1)));

            }
    int x = fattRic(-5);
    }

- : unit/2 = ()
#
```

`progSem` $\Longrightarrow$

```
- : unit/2 = ()
#     let fp = FP(Value, Nat, "x") in
      let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
      let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
      let espr = Times(Val "x", espaux) in
      let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
      let bl = BlockP(ED,SeqS(aux1,aux2)) in
      let d1 = Pcd(Nat, "fattRic", fp, bl) in
      let d2 = Var(Int, "x", Apply("fattRic", AP(N (-5)))) in
      let d = SeqD(d1,d2) in
      let prog = Prog("-5fatt_ricors_nat", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorT "E61.1: Trasmission: Type not expected".
#
```

Fattoriale - Esempi (2)

Programma che calcola un'applicazione della funzione fattoriale di un *naturale* all'intero -5.

```
let fp = FP(Value, Nat, "x") in
let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
let espr = Times(Val "x", espaux) in
let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
let bl = BlockP(ED,SeqS(aux1,aux2)) in
let d1 = Pcd(Nat, "fattRic", fp, bl) in
let d2 = Var(Int, "x", Apply("fattRic", AP(N (-5)))) in
let d = SeqD(d1,d2) in
let prog = Prog("-5fatt_ricors_nat", Block(d,UnL ES)) in
printProg prog
```

$\Downarrow$ `printProg`

```
Program -5fatt_ricors_nat{
    nat fattRic(value nat x){
            if (x == 0) return 1;
            if ((x == 0) == false) return (x * fattRic((x - 1)));

            }
    int x = fattRic(-5);
    }

- : unit/2 = ()
#
```

`progSem` $\Longrightarrow$

```
- : unit/2 = ()
#     let fp = FP(Value, Nat, "x") in
      let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
      let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
      let espr = Times(Val "x", espaux) in
      let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
      let bl = BlockP(ED,SeqS(aux1,aux2)) in
      let d1 = Pcd(Nat, "fattRic", fp, bl) in
      let d2 = Var(Int, "x", Apply("fattRic", AP(N (-5)))) in
      let d = SeqD(d1,d2) in
      let prog = Prog("-5fatt_ricors_nat", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorT "E61.1: Trasmission: Type not expected".
#
```

Di nuovo, il programma restituisce un errore di trasmissione dei parametri (l'espressione `N` $-$ 5 ha tipo `int`)

Fattoriale - Esempi (3)

Programma che calcola l'applicazione della funzione (che calcola il) fattoriale di un *naturale* al naturale 5.

Fattoriale - Esempi (3)

Programma che calcola l'applicazione della funzione (che calcola il) fattoriale di un *naturale* al naturale 5.

```
let fp = FP(Value, Nat, "x") in
let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
let espr = Times(Val "x", espaux) in
let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
let bl = BlockP(ED,SeqS(aux1,aux2)) in
let d1 = Pcd(Nat, "fattRic", fp, bl) in
let d2 = Var(Int, "x", Apply("fattRic", AP(N (5)))) in
let d = SeqD(d1,d2) in
let prog = Prog("-5fatt_ricors_nat", Block(d,UnL ES)) in
printProg prog
```

$\Downarrow$ `printProg`

```
Program -5fatt_ricors_nat{
    nat fattRic(value nat x){
            if (x == 0) return 1;
            if ((x == 0) == false) return (x * fattRic((x - 1)));

            }
    int x = fattRic(5);
    }

- : unit/2 = ()
#
```

$\xRightarrow{\texttt{progSem}}$

```
Stack:
>{-5fatt_ricors_nat,0,[x/(Mint,L6);
 fattRic/([nat::nat],$fattRic,[nat::nat],:fpar:,:cmd:,1$)],:cmdNext:,[N]}
]
Store:
[L0<-5,L1<-4,L2<-3,L3<-2,L4<-1,L5<-0,L6<-120]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Fattoriale - Esempi (3)

Programma che calcola l'applicazione della funzione (che calcola il) fattoriale di un *naturale* al naturale 5.

```
let fp = FP(Value, Nat, "x") in
let aux1 = IfT(Eq(Val "x", N 0), Return(N 1)) in
let espaux = Apply("fattRic", AP(Minus(Val "x", N 1))) in
let espr = Times(Val "x", espaux) in
let aux2 = IfT(Eq(Eq(Val "x", N 0), B False), Return(espr)) in
let bl = BlockP(ED,SeqS(aux1,aux2)) in
let d1 = Pcd(Nat, "fattRic", fp, bl) in
let d2 = Var(Int, "x", Apply("fattRic", AP(N (5)))) in
let d = SeqD(d1,d2) in
let prog = Prog("-5fatt_ricors_nat", Block(d,UnL ES)) in
printProg prog
```

$\Downarrow$ `printProg`

```
Program -5fatt_ricors_nat{
    nat fattRic(value nat x){
            if (x == 0) return 1;
            if ((x == 0) == false) return (x * fattRic((x - 1)));

            }
    int x = fattRic(5);
    }

- : unit/2 = ()
#
```

$\xRightarrow{\texttt{progSem}}$

```
Stack:
>{-5fatt_ricors_nat,0,[x/(Mint,L6);
 fattRic/([nat::nat],$fattRic,[nat::nat],:fpar:,:cmd:,1$)],:cmdNext:,[N]}
]
Store:
[L0<-5,L1<-4,L2<-3,L3<-2,L4<-1,L5<-0,L6<-120]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Il programma termina correttamente, restituendo il risultato esatto

Dichiarazioni con assegnamento - Esempi

Il seguente esempio mostra dichiarazioni (di variabili di tipo `nat` o `int`) con assegnamento, che vanno a buon fine.

```
let d1 = Var(Int, "x", N (-5)) in
let d2 = Var(Nat, "y", N (7)) in
let d3 = Var(Int, "z", Cast(Int, Val "y")) in
let d4 = Var(Int, "k", Val "y") in
let d = SeqD(SeqD(SeqD(d1,d2),d3),d4) in
let prog = Prog("assegnamento", Block(d,UnL ES)) in
printProg prog
```

printProg $\Longrightarrow$

```
Program assegnamento{
    int x = -5;
    nat y = 7;
    int z = Cast(int, y);
    int k = y;
    }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

```
Stack:
>{assegnamento,0,[k/(Mint,L3);
 z/(Mint,L2);
 y/(Mnat,L1);
 x/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<--5,L1<-7,L2<-7,L3<-7]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Dichiarazioni con assegnamento - Esempi

Il seguente esempio mostra dichiarazioni (di variabili di tipo `nat` o `int`) con assegnamento, che vanno a buon fine.

```
let d1 = Var(Int, "x", N (-5)) in
let d2 = Var(Nat, "y", N (7)) in
let d3 = Var(Int, "z", Cast(Int, Val "y")) in
let d4 = Var(Int, "k", Val "y") in
let d = SeqD(SeqD(SeqD(d1,d2),d3),d4) in
let prog = Prog("assegnamento", Block(d,UnL ES)) in
printProg prog
```

$\xrightarrow{\texttt{printProg}}$

```
Program assegnamento{
    int x = -5;
    nat y = 7;
    int z = Cast(int, y);
    int k = y;
    }

- : unit/2 = ()
#
```

$\xrightarrow{\texttt{progSem}}$

```
Stack:
>{assegnamento,0,[k/(Mint,L3);
 z/(Mint,L2);
 y/(Mnat,L1);
 x/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<--5,L1<-7,L2<-7,L3<-7]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Più interessante è la seguente dichiarazione errata:

```
let d = Var(Nat, "y", N (-5)) in
let prog = Prog("IllegalDcl", Block(d,UnL ES)) in
printProg prog
```

$\Longrightarrow$

```
# let d = Var(Nat, "y", N (-5)) in
      let prog = Prog("IllegalDcl", Block(d,UnL ES)) in
      printProg prog;;

Program IllegalDcl{
     nat y = -5;      }

- : unit/2 = ()
# let d = Var(Nat, "y", N (-5)) in
      let prog = Prog("IllegalDcl", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorI ("E4: dclSem", "y").
#
```

Cast - Esempi (1)

Esempi di uso di `Cast` nelle dichiarazioni:

```
let d1 = Var(Int, "x1", N 5) in
let esp = Cast(Int, N 5) in
let d2 = Var(Int, "x2", esp) in
let d = SeqD(d1,d2) in
let prog = Prog("CastEquiv", Block(d,UnL ES)) in
printProg prog
```

$\xrightarrow{\texttt{printProg}}$

```
Program CastEquiv{
    int x1 = 5;
    int x2 = Cast(int, 5);
    }

- : unit/2 = ()
#
```

$\xrightarrow{\texttt{progSem}}$

```
Stack:
>{CastEquiv,0,[x2/(Mint,L1);
 x1/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-5,L1<-5]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Cast - Esempi (1)

Esempi di uso di `Cast` nelle dichiarazioni:

```
let d1 = Var(Int, "x1", N 5) in
let esp = Cast(Int, N 5) in
let d2 = Var(Int, "x2", esp) in
let d = SeqD(d1,d2) in
let prog = Prog("CastEquiv", Block(d,UnL ES)) in
printProg prog
```

$\xrightarrow{\texttt{printProg}}$

```
Program CastEquiv{
     int x1 = 5;
     int x2 = Cast(int, 5);
     }

- : unit/2 = ()
#
```

$\xrightarrow{\texttt{progSem}}$

```
Stack:
>{CastEquiv,0,[x2/(Mint,L1);
 x1/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-5,L1<-5]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Più interessante è la seguente dichiarazione errata, ma con un cast eseguito correttamente:

```
let esp = Cast(Int, N 5) in
let d = Var(Nat,"y",esp) in
let prog = Prog("CastEquiv", Block(d,UnL ES)) in
printProg prog
```

$\Longrightarrow$

```
- : unit/2 = ()
#     let esp = Cast(Int, N 5) in
      let d = Var(Nat,"y",esp) in
      let prog = Prog("CastEquiv", Block(d,UnL ES)) in
      printProg prog;;

Program CastEquiv{
     nat y = Cast(int, 5);     }

- : unit/2 = ()
#     let esp = Cast(Int, N 5) in
      let d = Var(Nat,"y",esp) in
      let prog = Prog("CastEquiv", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorI ("E4: dclSem", "y").
#
```

Cast - Esempi (2)

Vediamo qualche esempio di cast errato:

```
let esp = Cast(Int, B True) in
let d = Var(Int, "x", esp) in
let prog = Prog("IllegalCast1", Block(d,UnL ES)) in
printProg prog
```

$\Longrightarrow$

```
Program IllegalCast1{
     int x = Cast(int, true);     }

- : unit/2 = ()
#     let esp = Cast(Int, B True) in
      let d = Var(Int, "x", esp) in
      let prog = Prog("IllegalCast1", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorE ("E801: expSem: - ", B True).
#
```

```
let esp = Cast(Nat, N (-5)) in
let d = Var(Nat, "y2", esp) in
let prog = Prog("IllegalCast2", Block(d,UnL ES)) in
printProg prog
```

$\Longrightarrow$

```
Program IllegalCast2{
     nat y2 = Cast(nat, -5);     }

- : unit/2 = ()
#     let esp = Cast(Nat, N (-5)) in
      let d = Var(Nat, "y2", esp) in
      let prog = Prog("IllegalCast2", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorE ("E802: expSem: -", N (-5)).
#
```

```
let esp = Cast(Bool, N 1) in
let d = Var(Bool, "b", esp) in
let prog = Prog("IllegalCast3", Block(d,UnL ES)) in
printProg prog
```

$\Longrightarrow$

```
Program IllegalCast3{
     bool b = Cast(bool, 1);     }

- : unit/2 = ()
#     let esp = Cast(Bool, N 1) in
      let d = Var(Bool, "b", esp) in
      let prog = Prog("IllegalCast3", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorE ("E800: expSem: - ", N 1).
#
```

Operazioni aritmetiche - Esempi (1)

Operazioni aritmetiche con operandi di tipo omogeneo

```
let op1 = Plus(N 3, N 2) in
let d1 = Var(Nat, "x1", op1) in
let op2 = Minus(N 3, N 2) in
let d2 = Var(Nat, "x2", op2) in
let op3 = Times(N 3, N 2) in
let d3 = Var(Nat, "x3", op3) in
let d4 = Var(Int, "y1", op1) in
let d5 = Var(Int, "y2", op2) in
let d6 = Var(Int, "y3", op3) in
let d = SeqD(SeqD(SeqD(SeqD(SeqD(d1,d2),d3),d4),d5),d6) in
let prog = Prog("OperandiOmogenei", Block(d,UnL ES)) in
printProg prog
```

printProg ⟹

```
Program OperandiOmogenei{
    nat x1 = (3 + 2);
    nat x2 = (3 - 2);
    nat x3 = (3 * 2);
    int y1 = (3 + 2);
    int y2 = (3 - 2);
    int y3 = (3 * 2);
    }

- : unit/2 = ()
#
```

progSem ⟹

```
Stack:
>{OperandiOmogenei,0,[y3/(Mint,L5);
 y2/(Mint,L4);
 y1/(Mint,L3);
 x3/(Mnat,L2);
 x2/(Mnat,L1);
 x1/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-5,L1<-1,L2<-6,L3<-5,L4<-1,L5<-6]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Operazioni aritmetiche - Esempi (1)

Operazioni aritmetiche con operandi di tipo omogeneo

```
let op1 = Plus(N 3, N 2) in
let d1 = Var(Nat, "x1", op1) in
let op2 = Minus(N 3, N 2) in
let d2 = Var(Nat, "x2", op2) in
let op3 = Times(N 3, N 2) in
let d3 = Var(Nat, "x3", op3) in
let d4 = Var(Int, "y1", op1) in
let d5 = Var(Int, "y2", op2) in
let d6 = Var(Int, "y3", op3) in
let d = SeqD(SeqD(SeqD(SeqD(SeqD(d1,d2),d3),d4),d5),d6) in
let prog = Prog("OperandiOmogenei", Block(d,UnL ES)) in
printProg prog
```

printProg $\Longrightarrow$

```
Program OperandiOmogenei{
    nat x1 = (3 + 2);
    nat x2 = (3 - 2);
    nat x3 = (3 * 2);
    int y1 = (3 + 2);
    int y2 = (3 - 2);
    int y3 = (3 * 2);
    }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

```
Stack:
>{OperandiOmogenei,0,[y3/(Mint,L5);
 y2/(Mint,L4);
 y1/(Mint,L3);
 x3/(Mnat,L2);
 x2/(Mnat,L1);
 x1/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-5,L1<-1,L2<-6,L3<-5,L4<-1,L5<-6]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Operandi di tipo non omogeneo: un primo esempio

```
let d1 = Var(Nat, "x", N 3) in
let d2 = Var(Int, "y", N 5) in
let op = Plus(Val "x", Val "y") in
let d3 = Var(Nat, "z", op) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("opNonH_err", Block(d,UnL ES)) in
printProg prog
```

printProg $\Longrightarrow$

```
Program opNonH_err{
    nat x = 3;
    int y = 5;
    nat z = (x + y);
    }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

Operazioni aritmetiche - Esempi (1)

Operazioni aritmetiche con operandi di tipo omogeneo

```
let op1 = Plus(N 3, N 2) in
let d1 = Var(Nat, "x1", op1) in
let op2 = Minus(N 3, N 2) in
let d2 = Var(Nat, "x2", op2) in
let op3 = Times(N 3, N 2) in
let d3 = Var(Nat, "x3", op3) in
let d4 = Var(Int, "y1", op1) in
let d5 = Var(Int, "y2", op2) in
let d6 = Var(Int, "y3", op3) in
let d = SeqD(SeqD(SeqD(SeqD(SeqD(d1,d2),d3),d4),d5),d6) in
let prog = Prog("OperandiOmogenei", Block(d,UnL ES)) in
printProg prog
```

printProg $\Longrightarrow$

```
Program OperandiOmogenei{
    nat x1 = (3 + 2);
    nat x2 = (3 - 2);
    nat x3 = (3 * 2);
    int y1 = (3 + 2);
    int y2 = (3 - 2);
    int y3 = (3 * 2);
    }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

```
Stack:
>{OperandiOmogenei,0,[y3/(Mint,L5);
 y2/(Mint,L4);
 y1/(Mint,L3);
 x3/(Mnat,L2);
 x2/(Mnat,L1);
 x1/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-5,L1<-1,L2<-6,L3<-5,L4<-1,L5<-6]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Operandi di tipo non omogeneo: un primo esempio

```
let d1 = Var(Nat, "x", N 3) in
let d2 = Var(Int, "y", N 5) in
let op = Plus(Val "x", Val "y") in
let d3 = Var(Nat, "z", op) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("opNonH_err", Block(d,UnL ES)) in
printProg prog
```

printProg $\Longrightarrow$

```
Program opNonH_err{
    nat x = 3;
    int y = 5;
    nat z = (x + y);
    }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

```
Stack:
>{opNonH_err,0,[z/(Mnat,L2);
 y/(Mint,L1);
 x/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-5,L2<-8]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Operazioni aritmetiche - Esempi (1)

Operazioni aritmetiche con operandi di tipo omogeneo

```
let op1 = Plus(N 3, N 2) in
let d1 = Var(Nat, "x1", op1) in
let op2 = Minus(N 3, N 2) in
let d2 = Var(Nat, "x2", op2) in
let op3 = Times(N 3, N 2) in
let d3 = Var(Nat, "x3", op3) in
let d4 = Var(Int, "y1", op1) in
let d5 = Var(Int, "y2", op2) in
let d6 = Var(Int, "y3", op3) in
let d = SeqD(SeqD(SeqD(SeqD(SeqD(d1,d2),d3),d4),d5),d6) in
let prog = Prog("OperandiOmogenei", Block(d,UnL ES)) in
printProg prog
```

$\xrightarrow{\texttt{printProg}}$

```
Program OperandiOmogenei{
    nat x1 = (3 + 2);
    nat x2 = (3 - 2);
    nat x3 = (3 * 2);
    int y1 = (3 + 2);
    int y2 = (3 - 2);
    int y3 = (3 * 2);
    }

- : unit/2 = ()
#
```

$\xrightarrow{\texttt{progSem}}$

```
Stack:
>{OperandiOmogenei,0,[y3/(Mint,L5);
 y2/(Mint,L4);
 y1/(Mint,L3);
 x3/(Mnat,L2);
 x2/(Mnat,L1);
 x1/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-5,L1<-1,L2<-6,L3<-5,L4<-1,L5<-6]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Operandi di tipo non omogeneo: un primo esempio

```
let d1 = Var(Nat, "x", N 3) in
let d2 = Var(Int, "y", N 5) in
let op = Plus(Val "x", Val "y") in
let d3 = Var(Nat, "z", op) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("opNonH_err", Block(d,UnL ES)) in
printProg prog
```

$\xrightarrow{\texttt{printProg}}$

```
Program opNonH_err{
    nat x = 3;
    int y = 5;
    nat z = (x + y);
    }

- : unit/2 = ()
#
```

$\xrightarrow{\texttt{progSem}}$

```
Stack:
>{opNonH_err,0,[z/(Mnat,L2);
 y/(Mint,L1);
 x/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-5,L2<-8]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

L'assegnamento non cause errore, in accordo alla semantica (è possibile fare downcast)

operazioni aritmetiche - Esempi (2)

Esempio di sottrazione con operandi di tipo non omogeneo:

```
let d1 = Var(Int, "x", N 5) in
let d2 = Var(Nat, "y", N 3) in
let d = SeqD(d1,d2) in
let op = Minus(Val "x", Val "y") in
let st = Upd(Val "x", op) in
let prog = Prog("minusNonH", Block(d, UnL st )) in
printProg prog
```

printProg $\Longrightarrow$

```
Program minusNonH{
    int x = 5;
    nat y = 3;
    x = (x - y);     }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

```
Stack:
>{minusNonH,0,[y/(Mnat,L1);
 x/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-2,L1<-3]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Operazioni aritmetiche - Esempi (3)

Vediamo un caso in cui non è possibile il downcast dell'operando di tipo più debole, rendendo quindi necessario l'upcast dell'altro operando.

```
let d1 = Var(Nat, "x", N 5) in
let d2 = Var(Int, "y", N(-3) ) in
let op = Plus(Val "x", Val "y") in
let d3 = Var(Nat, "z", op) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("opNonH_err", Block(d,UnL ES)) in
printProg prog
```

$\Longrightarrow$

Operazioni aritmetiche - Esempi (3)

Vediamo un caso in cui non è possibile il downcast dell'operando di tipo più debole, rendendo quindi necessario l'upcast dell'altro operando.

```
let d1 = Var(Nat, "x", N 5) in
let d2 = Var(Int, "y", N(-3) ) in
let op = Plus(Val "x", Val "y") in
let d3 = Var(Nat, "z", op) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("opNonH_err", Block(d,UnL ES)) in
printProg prog
```

$\Longrightarrow$

```
Program opNonH_err{
     nat x = 5;
     int y = -3;
     nat z = (x + y);
     }

- : unit/2 = ()
#     let d1 = Var(Nat, "x", N 5) in
      let d2 = Var(Int, "y", N(-3) ) in
      let op = Plus(Val "x", Val "y") in
      let d3 = Var(Nat, "z", op) in
      let d = SeqD(SeqD(d1,d2),d3) in
      let prog = Prog("opNonH_err", Block(d,UnL ES)) in
      progSem prog;;
Exception: TypeErrorI ("E4: dclSem", "z").
#
```

Osservazione

Il programma restituisce un errore di tipo. Ciò fa capire che stiamo applicando $+_{\texttt{int}}$, che ha segnatura $\left[\texttt{abs}\right]\,[\texttt{int}][::][\texttt{int}][::][\texttt{int}]$

Operazioni aritmetiche - Esempi (4)

Errore fastidioso con l'operazione di sottrazione:

```
let d1 = Var(Nat, "x", N 3) in
let d2 = Var(Int, "y", N 5) in
let op = Minus(Val "x", Val "y") in
let d3 = Var(Int, "z", op) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("errorMinus", Block(d, UnL ES)) in
printProg prog
```

$\Longrightarrow$

```
Program errorMinus{
     nat x = 3;
     int y = 5;
     int z = (x - y);
     }

- : unit/2 = ()
#      let d1 = Var(Nat, "x", N 3) in
       let d2 = Var(Int, "y", N 5) in
       let op = Minus(Val "x", Val "y") in
       let d3 = Var(Int, "z", op) in
       let d = SeqD(SeqD(d1,d2),d3) in
       let prog = Prog("errorMinus", Block(d, UnL ES)) in
       progSem prog;;
Exception: TypeErrorE ("E27: expSem - ", Minus (Val "x", Val "y")).
#
```

Osservazione

L'errore segue dal fatto che è possibile fare `Cast(nat, y)`, quindi l'operazione eseguita è $-_{\texttt{nat}}$, che però non è ben definita (solleva infatti E27). L'errore non si elimina applicando `Cast` all'intera espressione, bensì promuovendo `x`

Operazioni aritmetiche - Esempi (5)

Altra situazione poco piacevole:

Questo programma termina correttamente:

```
Program unpleasant1{
     nat x = 5;      x = (x - 1);      }

- : unit/2 = ()
#     let d = Var(Nat, "x", N 5) in
      let op = Minus(Val "x", N 1) in
      let st = Upd(Val "x", op) in
      let prog = Prog("unpleasant1", Block(d, UnL st)) in
      progSem prog;;

Stack:
>{unpleasant1,0,[x/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-4]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Operazioni aritmetiche - Esempi (5)

Altra situazione poco piacevole:

Questo programma termina correttamente:

```
Program unpleasant1{
     nat x = 5;      x = (x - 1);      }

- : unit/2 = ()
#      let d = Var(Nat, "x", N 5) in
       let op = Minus(Val "x", N 1) in
       let st = Upd(Val "x", op) in
       let prog = Prog("unpleasant1", Block(d, UnL st)) in
       progSem prog;;

Stack:
>{unpleasant1,0,[x/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-4]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Mentre questo restituisce un errore

Operazioni aritmetiche - Esempi (5)

Altra situazione poco piacevole:

Questo programma termina correttamente:

```
Program unpleasant1{
     nat x = 5;      x = (x - 1);      }

- : unit/2 = ()
#      let d = Var(Nat, "x", N 5) in
       let op = Minus(Val "x", N 1) in
       let st = Upd(Val "x", op) in
       let prog = Prog("unpleasant1", Block(d, UnL st)) in
       progSem prog;;

Stack:
>{unpleasant1,0,[x/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-4]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Mentre questo restituisce un errore

```
#      let d = Var(Nat, "x", N 5) in
       let op = Plus(Val "x", N (-1)) in
       let st = Upd(Val "x", op) in
       let prog = Prog("unpleasant1", Block(d, UnL st)) in
       printProg prog;;

Program unpleasant1{
     nat x = 5;      x = (x + -1);      }

- : unit/2 = ()
#      let d = Var(Nat, "x", N 5) in
       let op = Plus(Val "x", N (-1)) in
       let st = Upd(Val "x", op) in
       let prog = Prog("unpleasant1", Block(d, UnL st)) in
       progSem prog;;
Exception: TypeErrorS ("E39: stmSem", Upd (Val "x", Plus (Val "x", N (-1)))).
#
```

Operazioni aritmetiche - Esempi (5)

Altra situazione poco piacevole:

Questo programma termina correttamente:

```
Program unpleasant1{
     nat x = 5;      x = (x - 1);      }

- : unit/2 = ()
#      let d = Var(Nat, "x", N 5) in
       let op = Minus(Val "x", N 1) in
       let st = Upd(Val "x", op) in
       let prog = Prog("unpleasant1", Block(d, UnL st)) in
       progSem prog;;

Stack:
>{unpleasant1,0,[x/(Mnat,L0)],:cmdNext:,[N]}
]
Store:
[L0<-4]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Mentre questo restituisce un errore

```
#      let d = Var(Nat, "x", N 5) in
       let op = Plus(Val "x", N (-1)) in
       let st = Upd(Val "x", op) in
       let prog = Prog("unpleasant1", Block(d, UnL st)) in
       printProg prog;;

Program unpleasant1{
     nat x = 5;      x = (x + -1);      }

- : unit/2 = ()
#      let d = Var(Nat, "x", N 5) in
       let op = Plus(Val "x", N (-1)) in
       let st = Upd(Val "x", op) in
       let prog = Prog("unpleasant1", Block(d, UnL st)) in
       progSem prog;;
Exception: TypeErrorS ("E39: stmSem", Upd (Val "x", Plus (Val "x", N (-1)))).
#
```

L'errore E39 restituito è di non compatibilità tra il tipo della r-espressione con quello della l-espressione

Operazioni relazionali - Esempi

Esempio di operatore relazionale che confronta due operandi di tipo diverso ma compatibile:

```
let d1 = Var(Int, "x", N 3) in
let d2 = Var(Nat, "y", N 5) in
let oper = GT(Val "x", Val "y") in
let d3 = Var(Bool, "b", oper) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("GTHom", Block(d, UnL ES)) in
printProg prog
```

printProg $\Longrightarrow$

```
Program GTHom{
     int x = 3;
     nat y = 5;
     bool b = (x > y);
     }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

```
Stack:
>{GTHom,0,[b/(Mbool,L2);
 y/(Mnat,L1);
 x/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-5,L2<-false]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Operazioni relazionali - Esempi

Esempio di operatore relazionale che confronta due operandi di tipo diverso ma compatibile:

```
let d1 = Var(Int, "x", N 3) in
let d2 = Var(Nat, "y", N 5) in
let oper = GT(Val "x", Val "y") in
let d3 = Var(Bool, "b", oper) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("GTHom", Block(d, UnL ES)) in
printProg prog
```

printProg $\Longrightarrow$

```
Program GTHom{
    int x = 3;
    nat y = 5;
    bool b = (x > y);
    }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

```
Stack:
>{GTHom,0,[b/(Mbool,L2);
 y/(Mnat,L1);
 x/(Mint,L0)],:cmdNext:,[N]}
]
Store:
[L0<-3,L1<-5,L2<-false]


SUCCESSFUL_TERMINATION
- : unit/2 = ()
#
```

Esempio di operatore relazionale che confronta operandi di tipo non compatibile:

```
let d1 = Var(Int, "x", N 3) in
let d2 = Var(Bool, "y", B True) in
let oper = Eq(Val "x", Val "y") in
let d3 = Var(Bool, "b", oper) in
let d = SeqD(SeqD(d1,d2),d3) in
let prog = Prog("ConfErr", Block(d, UnL ES)) in
printProg prog
```

printProg $\Longrightarrow$

```
Program ConfErr{
    int x = 3;
    bool y = true;
    bool b = (x == y);
    }

- : unit/2 = ()
#
```

progSem $\Longrightarrow$

```
#     let d1 = Var(Int, "x", N 3) in
      let d2 = Var(Bool, "y", B True) in
      let oper = Eq(Val "x", Val "y") in
      let d3 = Var(Bool, "b", oper) in
      let d = SeqD(SeqD(d1,d2),d3) in
      let prog = Prog("ConfErr", Block(d, UnL ES)) in
      progSem prog;;
Exception: TypeErrorE ("E25': expSem - ", Eq (Val "x", Val "y")).
#
```

Sommario

`nat` come sottotipo di `int`: vantaggi

Vantaggio: un tipo in più per programmare. Come già visto, può evitare computazioni non terminanti.
È possibile sfruttare il nuovo sottotipo per organizzare meglio il progetto: il tipo `nat` potrebbe essere usato per esprimere un concetto diverso da `int`.

`nat` come sottotipo di `int`: vantaggi

Vantaggio: un tipo in più per programmare. Come già visto, può evitare computazioni non terminanti.
È possibile sfruttare il nuovo sottotipo per organizzare meglio il progetto: il tipo `nat` potrebbe essere usato per esprimere un concetto diverso da `int`.

Gli svantaggi, in entità, superano i vantaggi.

`nat` come sottotipo di `int`: svantaggi (1)

Svantaggi:

`nat` come sottotipo di `int`: svantaggi (1)

Svantaggi:

- A causa di:
  - Ciò che il tipo `nat` rappresenta concettualmente
  - Il sottotipo non è user-defined
  - `nat` è rappresentato in memoria come un `int`

  Le operazioni aritmetiche sono problematiche da trattare

`nat` come sottotipo di `int`: svantaggi (1)

Svantaggi:

- A causa di:
  - Ciò che il tipo `nat` rappresenta concettualmente
  - Il sottotipo non è user-defined
  - `nat` è rappresentato in memoria come un `int`

  Le operazioni aritmetiche sono problematiche da trattare
- È un tipo "da cui è facile uscire" con le usuali operazioni aritmetiche

`nat` come sottotipo di `int`: svantaggi (1)

Svantaggi:

- A causa di:
  - Ciò che il tipo `nat` rappresenta concettualmente
  - Il sottotipo non è user-defined
  - `nat` è rappresentato in memoria come un `int`

  Le operazioni aritmetiche sono problematiche da trattare
- È un tipo "da cui è facile uscire" con le usuali operazioni aritmetiche
- Rischia in molte occasioni di esporre il programmatore a diversi, e frustranti, errori di tipo

`nat` come sottotipo di `int`: svantaggi (2)

Confusione di ruoli: l'introduzione di `nat` parte da un'esigenza *del programmatore*, non del progettista di linguaggi!

`nat` come sottotipo di `int`: svantaggi (2)

Confusione di ruoli: l'introduzione di `nat` parte da un'esigenza *del programmatore*, non del progettista di linguaggi!

Il sottotipo `nat` risponde a una problematica (restringere l'insieme di valori ammissibili) che qualche programmatore potrebbe avere, e giudicare rilevante per il proprio progetto.
Essa andrebbe pertanto gestita da lui con strumenti che il linguaggio offre, con regole rigide e precise (controlli condizionali, tipi user-defined...)

`nat` come sottotipo di `int`: svantaggi (2)

Confusione di ruoli: l'introduzione di `nat` parte da un'esigenza *del programmatore*, non del progettista di linguaggi!

Il sottotipo `nat` risponde a una problematica (restringere l'insieme di valori ammissibili) che qualche programmatore potrebbe avere, e giudicare rilevante per il proprio progetto.
Essa andrebbe pertanto gestita da lui con strumenti che il linguaggio offre, con regole rigide e precise (controlli condizionali, tipi user-defined...)
Invece la presenza di `nat` a livello di linguaggio introduce cambiamenti che chiunque dovrà gestire, e che qualche altro programmatore riterrà inutili o sgraditi.

Riferimenti bibliografici

Maurizio Gabbrielli, Simone Martini

Linguaggi di Programmazione: Principi e Paradigmi

McGraw-Hill, Milano, 2011

Marco Bellia

Definizione di Small21: *Small21-Definizione5.pdf*

Giugno 2021

Marco Bellia

Materiale delle lezioni del corso: Lezione 9, *Lezione9.pdf*

Aprile 2021

Marco Bellia

Materiale delle lezioni del corso: Lezione 1, *Lezione1.pdf*

Febbraio 2021

Grazie per l'attenzione!