Linguaggi a memoria condivisa

L'idea è di aggiungere ad un linguaggio imperativo dei costrutti per rappresentare l'esecuzione parallela di statements.

Supponiamo di aggiungere a LW la produzione:

Stat::={Stat || Stat}

L'idea (modello più semplice possibile) è di permettere l'esecuzione di un passo del pezzo di sinistra o di quello di destra, indifferentemente (semantica di interleaving), quindi il modo più conveniente di descrivere la semantica è SOS a piccoli passi

$$\frac{(st_1, s) \rightarrow_r (st'_1, s')}{(\{st_1 \parallel st_2\}, s) \rightarrow_r (\{st'_1 \parallel st_2\}, s')} \qquad \frac{(st_2, s) \rightarrow_r (st'_2, s')}{(\{st_1 \parallel st_2\}, s) \rightarrow_r (\{st_1 \parallel st'_2\}, s')}$$

$$\frac{(st_1, s) \rightarrow_r (\varepsilon, s')}{(\{st_1 \parallel st_2\}, s) \rightarrow_r (st_2, s')} \qquad \frac{(st_2, s) \rightarrow_r (\varepsilon, s')}{(\{st_1 \parallel st_2\}, s) \rightarrow_r (st_1, s')}$$

Non determinismo

Consideriamo il frammento {x=7; y=x*x;||x=1;}

Supponiamo di essere in un ambiente e stato corretti per l'esecuzione di questo frammento (cioè in cui x e y sono variabili di tipo intero).

Usando le regole possiamo costruire le seguenti due catene di esecuzione (riscrittura a piccoli passi, per esercizio costruire gli alberi di ogni passo)

$(\{x=7; y=x*x;||x=1;\},s) \rightarrow_r (\{y=x*x;||x=1\},s[7/r(x)]) \rightarrow_r (y= x*x;, s[1/r(x)]) \rightarrow_r (\Box, s[1/r(x), 1/r(y)])$

$(\{x=7; y=x*x;||x=1;\},s) \rightarrow_r (x=7; y=x*x;,s[1/r(x)]) \rightarrow_r (y= x*x;, s[7/r(x)]) \rightarrow_r (\Box, s[7/r(x), 49/r(y)])$

Questo

Ribadisce il fatto che in presenza di parallelismo non si può evitare il non determinismo

Fa capire la necessità di introdurre costrutti per permettere di "fermare il mondo" durante l'esecuzione di pezzi di codice (ad esempio per evitare che variabili inizializzate per essere usate in espressioni successive vengano modificate prima di poter essere usate)

PARWHILE

Linguaggio per parallelismo basato su memoria condivisa (Omicki-Cries) PARallelWHILE. Lo adattiamo a LW (=LinguaggioWhile).

Partiamo da un sottoinsieme minimale di LW (non vogliamo tirarci dietro problemi di scoping, visibilità e tipaggio) e aggiungiamo i costrutti per il parallelismo

Esecuzione parallela come visto prima

Stat ::= Id = NExp; | **while** (BExp) {Stats} | {PStats}|
if (BExp) {Stats} **else** {Stats} | **skip** | **?**(Id) | **!**(NExp)
| ⟨Stats⟩ | **await** BExp;

Stats ::= Stat | Stat Stats. PStats ::= **nil** | Stats || PStats.

NExp ::= Id | NExp + NExp | …

BExp ::= NExp Rel NExp | BExp BConn BExp | **not** Bexp | true |false

Rel ::= = | < | >. BConn ::= **and** | **or** | **imp**.

Ci sono solo variabili intere, tutte sono dichiarate (e inizializzate a 0), così non ci sono problemi di correttezza statica.

Esecuzione atomica, cioè ininterrompibile di una sequenza

Meccanismo per la collaborazione: permette di attendere che altri processi terminino la loro attività

SOS Small Steps

Regole mutuate da LW (con gli ovvi adattamenti, eg r è una qualsiasi funzione totale ed iniettiva da L_{Id}(LW) in Loc)

$$\frac{(st, s) \to_r (\epsilon, s')}{(st\ sts, s) \to_r (sts, s')} \qquad \frac{(st, s) \to_r (st', s')}{(st\ sts, s) \to_r (st'\ sts, s')}$$ concatenazione di statements

$$\frac{(e,s) \to_r (e',s')}{(x=e;,s) \to_r (x=e';,s')} \qquad \frac{}{(x=v;,(i,o,m)) \to_r (\epsilon,(i,o,m[v/r(x)]))}$$ assegnazione

$$\frac{}{(\textbf{while}\ (e)\ \{sts\}, s) \to_r (\textbf{if}\ (e)\ \{sts\ \textbf{while}\ (e)\ \{sts\}\}\ \textbf{else}\ \{\textbf{skip};\}, s)}$$ while

$$\frac{(e, s) \to_r (e', s')}{(\textbf{if}\ (e)\ \{sts\}\ \textbf{else}\ \{sts'\}, s) \to_r (\textbf{if}\ (e')\ \{sts\}\ \textbf{else}\ \{sts'\}, s')}$$ if then else

$$\frac{}{(\textbf{if}\ (\text{false})\ \{sts\}\ \textbf{else}\ \{sts'\}, s) \to_r (sts', s)} \qquad \frac{}{(\textbf{if}\ (\text{true})\ \{sts\}\ \textbf{else}\ \{sts'\}, s) \to_r (sts, s)}$$

$$\frac{}{(\textbf{skip};, s) \to_r (\epsilon, s)}$$ skip

$$\frac{}{(\textbf{?}x;,(v\ i,o,m)) \to_r (\epsilon,(i,o,m[v/r(x)]))}$$ read

$$\frac{(e,s) \to_r (e',s')}{(\textbf{!}e;,s) \to_r (\textbf{!}e';,s')} \qquad \frac{}{(\textbf{!}v;,(i,o,m)) \to_r (\epsilon,(i,v\ o,m))}$$ write

più tutte le regole per le operazioni base (che sono lasciate per esercizio)

Regole per Interleaving

Idea intuitiva e centrale

$$\frac{(\{sts_i\}, s) \rightarrow_r (\{sts'_i\}, s')}{(\{sts_1 \| \ldots \| sts_{i-1} \| sts_i \| sts_{i+1} \| \ldots \| sts_n \| \mathbf{nil}\}, s) \rightarrow_r (\{sts_1 \| \ldots \| sts_{i-1} \| sts'_i \| sts_{i+1} \| \ldots \| sts_n \| \mathbf{nil}\}, s')}$$

$$\frac{(sts, s) \rightarrow_r (sts', s')}{(\{sts \| psts\}, s) \rightarrow_r (\{sts' \| psts\}, s')}$$

può muovere il primo processo

$$\frac{(sts, s) \rightarrow_r (\square, s')}{(\{sts \| psts\}, s) \rightarrow_r (\{psts\}, s')}$$

se si esaurisce il primo lo elimino

$$\frac{(\{psts\}, s) \rightarrow_r (\{psts'\}, s')}{(\{sts \| psts\}, s) \rightarrow_r (\{sts \| psts'\}, s')}$$

può muovere uno degli altri processi

$$\frac{}{(\{\mathbf{nil}\}, s) \rightarrow_r (\square, s)}$$

quando ho elaborato tutto ho finito

$$\frac{(z=3;,(i,o,m)) \rightarrow_r (\square,(i,o,m[3/r(z)]))}{(\{z=3;\|\mathbf{nil}\},(i,o,m)) \rightarrow_r (\{\mathbf{nil}\},(i,o,m[3/r(z)]))}$$

$$\frac{(\{z=3;\|\mathbf{nil}\},(i,o,m)) \rightarrow_r (\{\mathbf{nil}\},(i,o,m[3/r(z)]))}{(\{y=2;\|z=3;\|\mathbf{nil}\},(i,o,m)) \rightarrow_r (\{y=2;\|\mathbf{nil}\},(i,o,m[3/r(z)]))}$$

$$\frac{(\{y=2;\|z=3;\|\mathbf{nil}\},(i,o,m)) \rightarrow_r (\{y=2;\|\mathbf{nil}\},(i,o,m[3/r(z)]))}{(\{x=1;\|y=2;\|z=3;\|\mathbf{nil}\},(i,o,m)) \rightarrow_r (\{x=1;\|y=2;\|\mathbf{nil}\},(i,o,m[3/r(z)]))}$$

Formalizzazione che rispetta la struttura della grammatica

Regole per Sincronizzazione

Sincronizzazione:
Finché b è falsa non faccio niente, ovvero se b è vera passo al prossimo comando
Primo tentativo di regola (ispirandosi al **while**)

$$\frac{}{(\textbf{await } b;, s) \rightarrow_r (\textbf{if } (b)\{\textbf{skip};\} \textbf{ else } \{\textbf{await } b\}, s)}$$

Problema:
In questo modo se b è falsa non è vero che non faccio niente, faccio delle riscritture che alla fine mi porteranno al punto di partenza.
Secondo tentativo:
Riscrivo la guardia

$$\frac{(b, s) \rightarrow_r (b',s')}{(\textbf{await } b;, s) \rightarrow_r (\textbf{await } b';, s')}$$

Se arrivo a vero ho finito

$$\frac{}{(\textbf{await true};, s) \rightarrow_r (\square, s)}$$

Se arrivo a falso, non ho più nessuna regola per andare avanti.
Se c'è un altro percorso di valutazione che porta a vero, lo troverò (non determinismo) in un altro albero.
Altrimenti non riuscirò a costruire un albero di valutazione

Regole per Esecuzione Atomica

Esecuzione atomica

Non possiamo seguire il solito stile (ogni piccolo passo di una sottoespressione induce un piccolo passo dell'espressione complessiva), perché lo spirito di questo costrutto è proprio di trasformare una sequenza di piccoli passi di esecuzione portata fino a compimento in un singolo piccolo passo dell'espressione

$$\frac{(sts, s) \rightarrow_r^* (\square, s')}{(\{sts\}; s) \rightarrow_r (\square; s')}$$

$$\frac{\dfrac{\dfrac{}{(7*x,s) \rightarrow_r (7*7,s)}}{(y=7*x;,s) \rightarrow_r (y=7*7;,s)} \quad \dfrac{\dfrac{\dfrac{}{(7*7,s) \rightarrow_r (49,s)}}{(y=7*7;,s) \rightarrow_r (y=49;,s)} \quad \dfrac{\dfrac{}{(y=49;,s) \rightarrow_r (\square,s')}}{(y=49;,s) \rightarrow_r^* (\square,s')}}{(y=7*7;,s) \rightarrow_r^* (\square,s')}}{\dfrac{\dfrac{\dfrac{}{(x*x,s) \rightarrow_r (7*x,s)}}{(y=x*x;,s) \rightarrow_r (y=7*x;,s)} \quad (y=7*x;,s) \rightarrow_r^* (\square,s')}{(y=x*x;,s) \rightarrow_r^* (\square,s')} \quad \begin{array}{l} s = (i,o,m[7/r(x)]) \\ s' = (i,o,m[7/r(x),49/r(y)]) \end{array}}$$

$$\frac{}{(x=7;y=x*x;,(i,o,m)) \rightarrow_r (y=x*x;,(i,o,m[7/r(x)]))}$$

$$(x=7; y=x*x;,(i,o,m)) \rightarrow_r^* (\square, (i,o,m[7/r(x),49/r(y)]))$$

$$(\{x=7; y=x*x;\square\},(i,o,m)) \rightarrow_r (\square,(i,o,m[7/r(x),49/r(y)]))$$

$$(\{\{x=7; y=x*x;\square\}\|x=1\|\mathbf{nil}\},(i,o,m)) \rightarrow_r (\{x=1;\|\mathbf{nil}\},(i,o,m\,[7/r(x),49/r(y)]))$$

Semantica di un programma PARWHILE

Cerchiamo di capire quale semantica può risultare dalle nostre regole.

esecuzioni non terminanti (riscrittura infinita), ad esempio generati dal while (come nel caso non parallelo)

esecuzioni con risultato deadlock globale (la configurazione non è della forma ($\square$,s) ma non c'è nessuna regola che si applichi:

x=0;{**await** x>0;||**nil**}

esecuzioni terminanti (riscrittura finita che termina in una configurazione della forma ($\square$,s)),

(quelli che funzionano nel caso non parallelo o)

{x=0;|| **await** x>0;x=x-1|| x=1;|| **nil**}

Non determinismo: dallo stesso termine possono partire tutte e 3 i tipi di riscrittura

Semantica **Composizionale**???

Da quanto visto, la semantica di un comando dovrebbe essere non più una funzione da stato in stato ma un **insieme** di tali funzioni.

Allora che senso ha la composizionalità?

In che senso ⟦st sts⟧ = "composizione di" ⟦st⟧ e ⟦sts⟧ ???

La composizione non è banale (anzi è così complicata da non aiutare la comprensione)

In altre parole, in questo caso abbiamo rinunciato ad un requisito che avevamo imposto fin dall'inizio.

L'uso del metodo formale permette di capire esattamente in che senso "programmi concorrenti" sono intrinsecamente più complicati di programmi sequenziali.