Trabajo Práctico de Programación Lógica

PLP

Fecha de entrega: 25 de octubre

1. Introducción

En este TP usaremos un lenguaje - que es un fragmento de cálculo llamado CCS - para representar procesos no determinísticos. El lenguaje a utilizar es el siguiente (para mayor simplicidad, no admitimos ciclos).

$$P, Q ::= \mathbf{0} \mid \mu.P \mid P + Q$$

Donde $\mu \in \mathsf{Act} \cup \{\tau\}$, Act es un conjunto de acciones que producen salidas (etiquetas a, b, $c \ldots$) y τ denota una acción interna (equivalente a una transición λ en Teoría de Lenguajes). 0 representa un proceso ya terminado (no puede realizar ninguna acción). El operador "." indica la sucesión de una acción y un proceso que la continúa. El operador "+" indica una elección entre dos procesos, pudiendo ejecutar uno o el otro.

Otra posible representación de estos procesos es mediante autómatas. Por ejemplo:

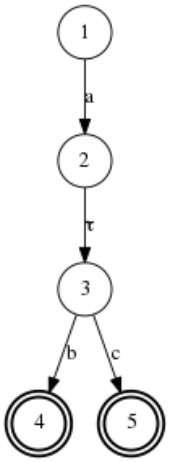

Figura 1: a.τ.(b.0 + c.0)

Representación utilizada

Representaremos el operador "." con el functor infijo "∗", el "+" con el "+", el proceso 0 con el número 0, y las acciones con átomos, siendo el átomo `tau` la representación de las transiciones internas.

Por ejemplo, el proceso $a.\tau.(b.0 + c.0)$ se representa como `a*tau*(b*0 + c*0)`.

2. Ejercicios

Diseñar e implementar en Prolog los siguientes predicados.

1. `acciones(+Proceso,-Acciones)`, donde `Acciones` es el conjunto de acciones que aparecen en `Proceso`.

   El conjunto de acciones que aparecen en un proceso P se define como:

$$\begin{aligned} \texttt{acciones}(\mathbf{0}) &= \emptyset \\ \texttt{acciones}(\tau.P) &= \texttt{acciones}(P) \\ \texttt{acciones}(\mu.P) &= \{\mu\} \cup \texttt{acciones}(P) \text{ si } \mu \neq \tau \\ \texttt{acciones}(P+Q) &= \texttt{acciones}(P) \cup \texttt{acciones}(Q) \end{aligned}$$

2. `reduce(+Proceso1,?Accion,?Proceso2)`, que indica que `Proceso1` reduce en un paso a `Proceso2` mediante la acción `Accion` $\in \mathsf{Act} \cup \{\tau\}$.

   La semántica operacional está definida de la siguiente manera:

$$\mu.P \xrightarrow{\mu} P \qquad \frac{P \xrightarrow{\mu} P'}{P+Q \xrightarrow{\mu} P'} \qquad \frac{Q \xrightarrow{\mu} Q'}{P+Q \xrightarrow{\mu} Q'}$$

3. `reduceLista(+Proceso1,?Cadena,?Proceso2)`, que es verdadero si `Cadena` es una lista de acciones con salida (es decir, distintas de τ) y `Proceso2` es el proceso que queda luego de reducir `Proceso1` produciendo las salidas de `Cadena` en el orden correspondiente.

   Para esto pueden realizarse acciones τ de ser necesario. Al no producir salida alguna, estas acciones no consumen elementos de la cadena. Por ejemplo:

```
?- reduceLista(0,S,Q).
S = [], Q = 0 ;
false.

?- reduceLista(c * ((a*0) + (b * tau * 0)), S, Q).
S = [], Q = (c* (a*0+b*tau*0)) ;
S = [c], Q = (a*0+b*tau*0) ;
S = [c, a], Q = 0 ;
S = [c, b], Q = (tau*0) ;
S = [c, b], Q = 0 ;
false.
```

   Notación: escribiremos $P \stackrel{S}{\Longrightarrow} Q$ en los casos en que `reduceLista(P,S,Q)` sea verdadero. Si no tenemos en cuenta las acciones τ, podemos decir que $P \stackrel{\mu_0 \cdots \mu_n}{\Longrightarrow} Q$ cuando $P \xrightarrow{\mu_0} P_0 \ldots \xrightarrow{\mu_n} P_n = Q$.

   Escribimos $P \stackrel{S}{\Longrightarrow}$ cuando existe Q tal que $P \stackrel{S}{\Longrightarrow} Q$.

4. `trazas(+Proceso, -Cadenas)`, que instancia en `Cadenas` el conjunto de cadenas que el proceso `Proceso` puede producir (siempre ignorando las acciones τ). Es decir, las cadenas S tales que `Proceso`$\stackrel{S}{\Longrightarrow}$. Por ejemplo:

```
?- trazas((a*0) + (b * 0), Trazas).
Trazas = [[], [a], [b]].

?- trazas(c * ((a*0) + (b * tau * 0)), Trazas).
Trazas = [[], [c], [c, a], [c, b]].
```

5. `residuo(+X,+Cadena,-Qs)`, que instancia en `Qs` el residuo de `X` luego de ejecutar la cadena `Cadena`. `X` puede ser un proceso o una lista de procesos (vista como conjunto).

   El residuo de un proceso P luego de la ejecución S es el conjunto de procesos definido como:

$$(P \text{ after } S) = \{Q \mid P \stackrel{S}{\Longrightarrow} Q\}$$

   Esta noción se generaliza a conjuntos de procesos de la siguiente manera:

$$(Ps \text{ after } S) = \bigcup_{P \in Ps} (P \text{ after } S)$$

   Ejemplos:

```
?- residuo((a*0+b*tau*0+b*c*0),[b],Q).
Q = [0, c*0, tau*0].

?- residuo((a*0+b*tau*0+b*c*0),[c],Q).
Q = [].

?- residuo([(a*0+b*tau*0+b*c*0),(b*a*c*0)],[b],Q).
Q = [0, c*0, tau*0, a*c*0].

?- residuo([(a*0+b*tau*0+b*c*0),(b*a*c*0+b*c*0)],[b],Q).
Q = [0, tau*0, a*c*0, c*0].
```

6. `must(+X,+L)`

   Un proceso P realiza una acción en el conjunto $L \subseteq \mathsf{Act}$ (con L finito), escrito como P must L, si se cumple que $\forall Q(P \stackrel{[\,]}{\Longrightarrow} Q \supset \exists \alpha \in L(Q \stackrel{[\alpha]}{\Longrightarrow}))$. Es decir, si todo Q perteneciente al residuo de P luego de realizar acciones internas puede realizar, como próxima acción no interna, una acción contenida en L.

   Esta noción se generaliza a conjuntos de procesos de la siguiente manera:

   Ps must L si $\forall P \in Ps.P$ must L.

   Por ejemplo:

```
?- must((a*0+b*0+b*c*0),[a]).
true.

?- must((tau*b*0+b*c*0),[b]).
true.

?- must((a*0+b*c*0),[a,b]).
true.
```

```
?- must((a*0+b*c*0),[c]).
false.

?- must([c*0,(b*c*0)],[c]).
false.

?- must([b*0,(a*0+b*c*0)],[b]).
true.
```

7. `puedeReemplazarA(+P, +Q)`, que es verdadero si `Q` puede reemplazar a `P`.

   Decimos que un proceso P puede ser reemplazado por otro Q, $P \preceq_{\mathsf{must}} Q$ si

   para todo $S \in \mathtt{trazas}(P) \cup \mathtt{trazas}(Q)$, $L \subseteq \mathtt{acciones}(P) \cup \mathtt{acciones}(Q)$ y L finito, se cumple que

   (P after S) must $L \supset$ (Q after S) must L.

   Sugerencia: escribir primero la negación de este predicado.

   En el siguiente ejemplo vamos a usar los números de los estados para hablar de los residuos:

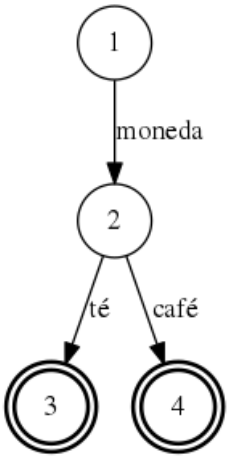

Figura 2: P= moneda.(te.0 + cafe.0)

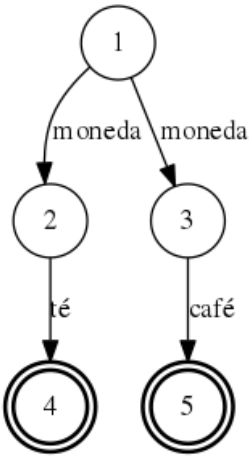

Figura 3: Q= moneda.te.0+moneda.cafe.0

Notar que para la figura 2:
(P **after** moneda) = {2}
y {2} **must** {café}.

Sin embargo para la figura 3:
(Q **after** moneda) = {2, 3} pero
no vale que ({2, 3} **must** {café}).

Luego **Q** no puede reemplazar a **P**.

8. `equivalentes(+P, +Q)` que indica que ambos procesos son intercambiables.

3. Condiciones de aprobación

El principal objetivo de este trabajo es evaluar el correcto uso del lenguaje PROLOG de forma declarativa para resolver el problema planteado. El código debe estar *adecuadamente* comentado (es decir, los comentarios que escriban deben facilitar la lectura de los predicados, y no ser una traducción al castellano del código). También se debe explicitar cuáles de los argumentos de los predicados auxiliares deben estar instanciados usando `+`, `-` y `?`.

La entrega debe incluir casos de prueba en los que se ejecute al menos una vez cada predicado definido.

En el caso de los predicados que utilicen la técnica de *generate and test*, deberán indicarlo en los comentarios.

4. Pautas de entrega

Se debe entregar el código impreso con la implementación de los predicados pedidos y los tests correspondientes. Además, se debe enviar un e-mail conteniendo el código fuente en Prolog a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PL]` seguido inmediatamente del nombre del grupo.
- El código Prolog debe acompañar el e-mail en forma de archivo adjunto.

El código debe poder ser ejecutado en SWI-Prolog. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado. Los objetivos a evaluar en la implementación de los predicados son:

- corrección,
- declaratividad,
- reutilización de predicados previamente definidos,
- utilización de unificación, backtracking, generate and test y reversibilidad de los predicados que correspondan.
- **Importante**: salvo donde se indique lo contrario, los predicados no deben instanciar soluciones repetidas ni colgarse luego de devolver la última solución. Vale aclarar que no es necesario filtrar las soluciones repetidas si la repetición proviene de las características de la entrada.

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

5. Referencias y sugerencias

Como referencia se recomienda la bibliografía incluída en el sitio de la materia (ver sección *Bibliografía* → *Programación Lógica*).

Se recomienda que utilicen los predicados ISO y los de SWI-Prolog ya disponibles, siempre que sea posible. Recomendamos especialmente examinar los predicados y metapredicados que figuran en la sección *Cosas útiles* de la página de la materia. Pueden hallar la descripción de los mismos en la ayuda de `SWI-Prolog` (a la que acceden con el predicado `help`). También se puede acceder a la documentación online de SWI-Prolog.