Trabajo Práctico 2
Programación Lógica

Paradigmas ~~de Lenguajes~~ de Programación
2do cuatrimestre 2024

Fecha de entrega: martes 19 de noviembre

1. Introducción

Se quiere modelar el funcionamiento de ciertos procesos que escriben en buffers. Cada buffer contiene una cola con los datos que se escribieron en él. Al escribir un dato, este se agrega al final. Al leer, se quita el primer elemento de la cola. A su vez, los procesos pueden contener subprocesos.

Se espera que resuelvan los ejercicios y se expresen en Prolog de forma elemental, declarativa y clara.

Representación utilizada

Los procesos que elegimos modelar se representan con distinas expresiones de Prolog.

Por un lado, están las acciones básicas:

- El átomo `computar`, que no realiza cambios en ningún buffer.
- El término `escribir(B,E)`, donde `B` es el buffer y `E` es lo que se quiere escribir en él. Por ejemplo, `escribir(b1,'casa')`.
- El término `leer(B)`, que lee la primera escritura del buffer `B`, por lo que lo elimina de él. Por ejemplo, `leer(b1)`.

Y, por otro lado, tenemos procesos que pueden incluir varias acciones:

- El término `secuencia(P,Q)`, donde `P` y `Q` son subprocesos que se ejecutan en ese orden (i.e., primero `P` y después `Q`). Por ejemplo, `secuencia(escribir(a8,hola),escribir(a8,chau))`.
- El término `paralelo(P,Q)`, donde `P` y `Q` son subprocesos que se ejecutan en 'paralelo' (i.e., sus acciones finalmente se realizan secuencialmente, intercaladas de alguna manera siguiendo su orden relativo, pero en un orden general no determinado a priori: cualquiera de los dos subprocesos puede realizar su siguiente acción en cualquier momento). Por ejemplo,

```
paralelo(secuencia(escribir(1,'hola'),escribir(1,'chau')),
         secuencia(escribir(2,'hallo'),escribir(2,'tchüss'))).
```

2. Ejercicios

A continuación se indican los predicados solicitados para el problema.

Predicados básicos

1. `proceso(+P)` que identifica si un término es un proceso.

```
?- proceso(computar).
true.
```

2. `buffersUsados(+P,-BS)`, que instancia en `BS` los buffers utilizados por el proceso `P`, en una lista ordenada y sin repetidos.

```
?- buffersUsados(escribir(1, hola), BS).
BS = [1] ;
false.
```

Organización de procesos

3. `intercalar(+XS,+YS,?ZS)`, que intercala dos listas en una tercera, de todas las maneras posibles, manteniendo el orden relativo de los elementos en cada una de las listas originales.

```
?- intercalar([1,2,3],[4,5,6],I).
I = [1,2,3,4,5,6] ;
I = [4,5,6,1,2,3] ;
I = [1,4,2,5,3,6] ;
...
false.
```

4. `serializar(+P,?XS)`, que instancia en XS una lista de acciones básicas (cómputos, lecturas y escrituras) que puede realizar el proceso en cualquiera de sus posibles ejecuciones, en el orden en que se realizan (en este punto no importa si falla una lectura).

```
?- serializar(secuencia(computar,leer(2)),XS).
XS = [computar,leer(2)] ;
false.

?- serializar(paralelo(paralelo(leer(1),leer(2)),secuencia(leer(3),leer(4))),XS).
XS = [leer(1),leer(3),leer(2),leer(4)] ;
XS = [leer(3),leer(1),leer(4),leer(2)] ;
...
false.
```

Contenido de los buffers

5. `contenidoBuffer(+B,+ProcesoOLista,?L)`, que instancia en `L` el contenido del buffer `B` luego de realizar las acciones indicadas por un proceso o una serialización, respetando el orden en que se realizan las acciones.

   La longitud de `L` es cantidad de escrituras en `B` menos la cantidad de lecturas en `B`. Da false si se intenta leer cuando no hay nada escrito.

   Si `ProcesoOLista` es un proceso que tiene varias serializaciones posibles, instancia una lista de contenidos por cada serialización, pudiendo devolver soluciones repetidas si y solo si existen distintas serializaciones que leen los mismos contenidos. No genera soluciones para las serializaciones que intentan leer un buffer vacío.

```
?- contenidoBuffer(1,[escribir(1,pa),escribir(2,ma),escribir(1,hola),
                      computar,escribir(1,mundo),leer(1)],C).
C = [hola, mundo] ;
false.

?- contenidoBuffer(2,[escribir(1,pp),escribir(2,ala),escribir(1,ola),
                      computar,escribir(1,mundo),leer(1)],C).
C = [ala] ;
false.

?- contenidoBuffer(2,paralelo(escribir(2,sol),secuencia(escribir(1,agua),leer(1))),C).
C = [sol] ;
C = [sol] ;
C = [sol] ;
false.

?- contenidoBuffer(1,paralelo(escribir(2,sol),secuencia(escribir(1,agua),leer(1))),XS).
XS = [] ;
XS = [] ;
XS = [] ;
false.

?- contenidoBuffer(1,paralelo(leer(1),escribir(1,agua)),XS).
XS = [] ;
false.
```

6. `contenidoLeido(+ProcesoOLista,?Contenidos)`, que instancia en `Contenidos` la lista de contenidos que se van leyendo, en el orden en que se leen. `ProcesoOLista` puede ser un proceso o la serialización de un proceso. Si es un proceso que tiene varias serializaciones posibles, instancia una lista de contenidos por cada serialización, pudiendo devolver soluciones repetidas si y solo si existen distintas serializaciones que leen los mismos contenidos. No genera soluciones para las serializaciones que intentan leer un buffer vacío.

```
?- contenidoLeido(paralelo(secuencia(escribir(2,sol),leer(2)),
                  secuencia(escribir(1,agua),leer(1))),CS).
CS = [sol, agua] ;
CS = [sol, agua] ;
CS = [agua, sol] ;
CS = [sol, agua] ;
CS = [agua, sol] ;
CS = [agua, sol] ;
false.

?- contenidoLeido([escribir(1, agua), escribir(2, sol), leer(1), leer(1)],CS).
false.

?- contenidoLeido(paralelo(secuencia(escribir(2,sol),secuencia(leer(2),escribir(2,agua))),
                  secuencia(leer(2),computar)),CS).
CS = [sol, agua] ;
false.
```

Secuencias y procesos seguros

Llamamos *ejecución segura* a una lista de acciones básicas (lecturas, escrituras y cómputos) tal que, si se ejecutan las acciones en el orden en que aparecen en la lista, nunca se intenta leer un buffer vacío.

Un *proceso* es *seguro* si todas sus serializaciones son ejecuciones seguras, y cada vez que se divide en dos subprocesos paralelos, estos subprocesos no comparten buffers.

7. `esSeguro(+P)`, que determina si un proceso es seguro.

```
?- esSeguro(secuencia(leer(1),escribir(1,agua))).
false.

?- esSeguro(secuencia(escribir(1,agua),leer(1))).
true.

?- esSeguro(paralelo(escribir(1,sol),secuencia(escribir(1,agua),leer(1)))).
false.

?- esSeguro(paralelo(escribir(2,sol),secuencia(escribir(1,agua),leer(1)))).
true.
```

8. `ejecucionSegura(-XS,+BS,+CS)`, que instancia en XS todas las posibles ejecuciones seguras con buffers de BS y contenidos de CS. Se pide usar *Generate & Test*.

```
?- ejecucionSegura(L,[1,2],[a,b]).
L = [] ;
L = [computar] ;
L = [escribir(1, a)] ;
L = [escribir(1, b)] ;
L = [escribir(2, a)] ;
L = [escribir(2, b)] ;
L = [computar, computar] ;
L = [computar, escribir(1, a)] ;
L = [computar, escribir(1, b)] ;
L = [computar, escribir(2, a)] ;
L = [computar, escribir(2, b)] ;
L = [escribir(1, a), computar] ;
L = [escribir(1, b), computar] ;
L = [escribir(2, a), computar] ;
L = [escribir(2, b), computar] ;
L = [escribir(1, a), escribir(1, a)] ;
```

```
L = [escribir(1, b), escribir(1, a)] ;
L = [escribir(2, a), escribir(1, a)] ;
L = [escribir(2, b), escribir(1, a)] ;
L = [escribir(1, a), escribir(1, b)] ;
L = [escribir(1, b), escribir(1, b)] ;
L = [escribir(2, a), escribir(1, b)] ;
L = [escribir(2, b), escribir(1, b)] ;
L = [escribir(1, a), escribir(2, a)] ;
L = [escribir(1, b), escribir(2, a)] ;
L = [escribir(2, a), escribir(2, a)] ;
L = [escribir(2, b), escribir(2, a)] ;
L = [escribir(1, a), escribir(2, b)] ;
L = [escribir(1, b), escribir(2, b)] ;
L = [escribir(2, a), escribir(2, b)] ;
L = [escribir(2, b), escribir(2, b)] ;
L = [escribir(1, a), leer(1)] ;
...
false.
```

8.1. Analizar la reversibilidad de `XS`, justificando adecuadamente por qué el predicado se comporta como lo hace.

3. Pautas de Entrega

Importante: Se espera que la elaboración de este trabajo sea 100 % de los estudiantes del grupo que realiza la entrega. Así que, más allá de que pueden tomar información de lo visto en las clases o consultar información en la documentación de SWI-Prolog u otra disponible en Internet, no se podrán utilizar herramientas para generar parcial o totalmente en forma automática la resolución del TP (e.g., ChatGPT, Copilot, etc). En caso de detectarse esto, el trabajo será considerado como un plagio, por lo que será gestionado de la misma forma que se resuelven las copias en los parciales u otras instancias de evaluación.

Cada predicado asociado a los ejercicios debe contar con ejemplos (tests) que muestren que exhibe la funcionalidad solicitada. Además, se deberá subir subir el código fuente a la tarea respectiva en el Campus.

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
- Uso 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. 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.

4. Referencias y sugerencias

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

Se recomienda que, siempre que sea posible, utilicen los predicados ISO y los de SWI-Prolog ya disponibles. Recomendamos especialmente examinar los predicados y metapredicados que figuran en la sección *Otros* 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.