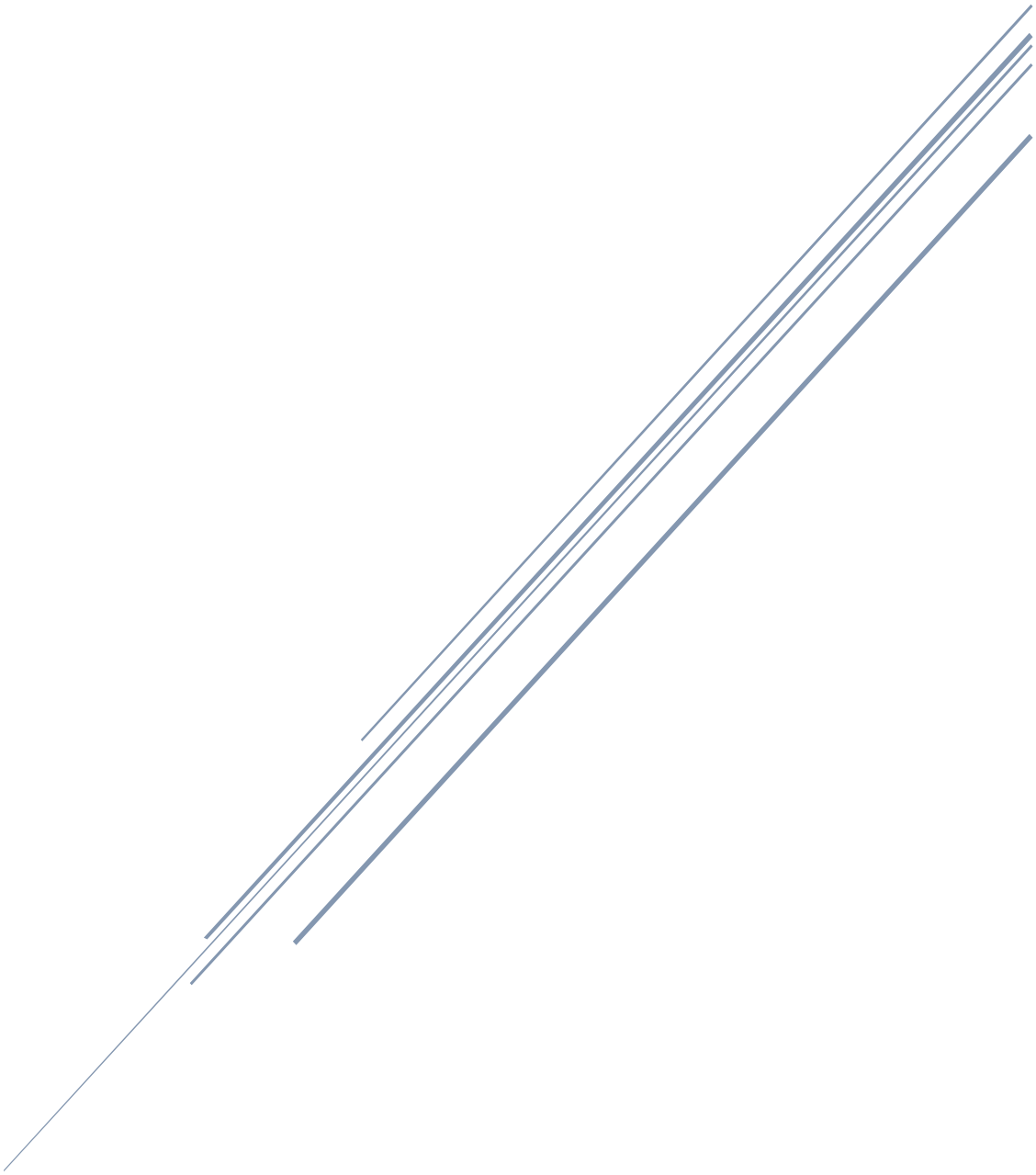
PRACTICA 3

LPM - Laboratorio de programación multiparadigma



MITSS
Sergi Sanz Carreres

Índice:

4.1. Monty Python Example

1.- ¿Qué devuelve la ejecución de la expresión vu-narrow bruja(X) =>* true .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow bruja(X) =>* true .
vu-narrow in BRUJA : bruja(X) =>* true .

No solution.
rewrites: 15 in 7ms cpu (10ms real) (1908 rewrites/second)
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **bruja(X)** a una instancia del término **true**, mientras que =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con X, que en este caso no hay.

2.- ¿Qué devuelve la ejecución de la expresión vu-narrow bruja(lola) =>* true .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow bruja(lola) =>* true .
vu-narrow in BRUJA : bruja(lola) =>* true .

No solution.
rewrites: 14 in 6ms cpu (6ms real) (2194 rewrites/second)
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **bruja(lola)** a una instancia del término **true**, mientras que =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con lola, que en este caso no hay.

3.- ¿Qué devuelve la ejecución de la expresión vu-narrow bruja(lola) ~>* true . si cambiamos la segunda regla de mismopeso para que compare pato y lola?

Para la realización de este ejercicio se ha tenido que sustituir la regla:

```
rl mismopeso(pato,jamon) => true [narrowing] .
```

Por la regla:

```
rl mismopeso(pato,lola) => true [narrowing] .
```

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow bruja(lola) =>* true .
vu-narrow in BRUJA : bruja(lola) =>* true .

Solution 1
rewrites: 17 in 5ms cpu (5ms real) (2996 rewrites/second)
state: true
accumulated substitution:
variant unifier:

Solution 2
rewrites: 19 in 6ms cpu (6ms real) (2992 rewrites/second)
state: true
accumulated substitution:
variant unifier:

Solution 3
rewrites: 21 in 7ms cpu (7ms real) (2985 rewrites/second)
state: true
accumulated substitution:
variant unifier:

Solution 4
rewrites: 23 in 7ms cpu (7ms real) (3060 rewrites/second)
state: true
accumulated substitution:
variant unifier:

Solution 5
rewrites: 25 in 7ms cpu (8ms real) (3127 rewrites/second)
state: true
accumulated substitution:
variant unifier:

No more solutions.
rewrites: 25 in 10ms cpu (11ms real) (2439 rewrites/second)
```

Donde se observa como =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con true.

4.2. Listas

1.- ¿Qué devuelve la ejecución de la expresión rewrite length(nil) .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> rewrite length(nil) .
rewrite in LISTAS : length(nil) .
rewrites: 1 in 0ms cpu (0ms real) (200000 rewrites/second)
result Zero: 0
```

Al aplicar reescritura se realiza un paso donde obtenemos la largaría de nil que es cero.

2. ¿Qué devuelve la ejecución de la expresión vu-narrow length(1 : 2 : 3 : 4 : nil) =>! N .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow length(1 : 2 : 3 : 4 : nil) =>! N .
vu-narrow in LISTAS : length(1 : 2 : 3 : 4 : nil) =>! N .

Solution 1
rewrites: 9 in 2ms cpu (3ms real) (3016 rewrites/second)
state: 4
accumulated substitution:
variant unifier:
N --> 4

No more solutions.
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **length(1:2:3:4:nil)** a una instancia del término **N**, mientras que =>! devuelve todos los posibles estados finales a los que se llega reescribiendo el término inicial que le pasamos y que hacen matching con el patrón de la derecha. Por tanto, obtendremos una solución en la que se nos indica que 4 estados hacen matching, obteniendo como resultado N -- > 4.

3. ¿Qué devuelve la ejecución de la expresión vu-narrow [1] length(NL) =>* 0 .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow [1] length(NL) =>* 0 .
vu-narrow [1] in LISTAS : length(NL) =>* 0 .

Solution 1
rewrites: 1 in 0ms cpu (0ms real) (6756 rewrites/second)
state: 0
accumulated substitution:
NL --> nil
variant unifier:
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **length(NL)** a una instancia del término **0**, en este caso se encuentra limitado a [1] solución. Mientras que =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con NL, que en este caso el resultado es nil.

4. ¿Qué devuelve la ejecución de la expresión vu-narrow [1] length(NL) =>* 10 .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow [1] length(NL) =>* 10 .
vu-narrow [1] in LISTAS : length(NL) =>* 10 .

Solution 1
rewrites: 40 in 5ms cpu (6ms real) (7330 rewrites/second)
state: 10
accumulated substitution:
NL --> @1:Nat : @2:Nat : @3:Nat : @4:Nat : @5:Nat : @6:Nat : @7:Nat : @8:Nat :
    @9:Nat : @10:Nat : nil
variant unifier:
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **length(NL)** a una instancia del término **10**, en este caso se encuentra limitado a [1] solución. Mientras que =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con NL, siendo en este caso el resultado es **NL -> @1:Nat : @2:Nat : @3:Nat : @4:Nat : @5:Nat : @6:Nat : @7:Nat : @8:Nat : @9:Nat : @10:Nat : nil**.

4.3. Suma y comparación

1.- ¿Qué devuelve la ejecución de la expresión vu-narrow [1] leq(sum(X,Y),1) =>* true .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow [1] leq(sum(X,Y),1) =>* true .
vu-narrow [1] in SUM-LEQ : leq(sum(X, Y), 1) =>* true .

Solution 1
rewrites: 3 in 1ms cpu (1ms real) (2759 rewrites/second)
state: true
accumulated substitution:
X --> 0
Y --> 0
variant unifier:
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **sum((X,Y), 1)** a una instancia del término **true**, en este caso se encuentra limitado a [1] solución. Mientras que =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con sum(X,Y),1), siendo en este caso el resultado X -> 0 e Y -> 0.

2.- ¿Qué devuelve la ejecución de la expresión vu-narrow [1] leq(sum(0,0),s(Y)) =>* true .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow [1] leq(sum(0,0),s(Y)) =>* true .
vu-narrow [1] in SUM-LEQ : leq(sum(0, 0), s Y) =>* true .

Solution 1
rewrites: 2 in 0ms cpu (0ms real) (2659 rewrites/second)
state: true
accumulated substitution:
Y --> %1:[Nat]
variant unifier:
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **sum((0,0), s(Y))** a una instancia del término **true**, en este caso se encuentra limitado a [1] solución. Mientras que =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con **sum((0,0), s(Y)),** siendo en este caso el resultado Y -> 0.

3.- ¿Qué devuelve la ejecución de la expresión vu-narrow [1] leq(sum(X,Y),s(X)) =>* true .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow [1] leq(sum(X,Y),s(X)) =>* true .
vu-narrow [1] in SUM-LEQ : leq(sum(X, Y), s X) =>* true .

Solution 1
rewrites: 3 in 0ms cpu (0ms real) (3562 rewrites/second)
state: true
accumulated substitution:
X --> 0
Y --> 0
variant unifier:
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **sum((X,Y), s(X))** a una instancia del término **true**, en este caso se encuentra limitado a [1] solución. Mientras que =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con **sum((X,Y), s(X)),** siendo en este caso el resultado X -> 0 e Y -> 0.

4.- ¿Qué devuelve la ejecución de la expresión vu-narrow [1] sum(X,Y) =:= 1 and leq(X,Y) =>* true .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow [1] sum(X,Y) =:= 1 and leq(X,Y) =>* true .
vu-narrow [1] in SUM-LEQ : leq(X, Y) and sum(X, Y) =:= 1 =>* true .

Solution 1
rewrites: 15 in 5ms cpu (5ms real) (2894 rewrites/second)
state: true
accumulated substitution:
X --> 0
Y --> 1
variant unifier:
```

En este caso la expresión **vu-narrow** se encarga de buscar algún camino de ejecución que lleve de una instancia del término **sum(X,Y) =:= 1 and leq(X,Y)** a una instancia del término **true**, estando limitado a [1] solución. Mientras que =>* devuelve todos los posibles estados a los que se llega con cero o más pasos de reescritura desde el estado inicial y que hagan matching con el patrón. Por tanto, se devolverá todos los posibles estados que hagan matching con **sum(X,Y) =:= 1 and leq(X,Y),** siendo en este caso el resultado X -> 0 e Y -> 1.

4.4. Resta

El código añadido para posibilitar esta funcionalidad es el siguiente:

```
op minus : Nat Nat -> [Nat] .
rl minus(X, 0) => X [narrowing] .
rl minus(s(X), s(Y) ) => minus(X, Y) [narrowing] .
```

1. ¿Qué devuelve la expresión vu-narrow [1] minus(s(0),s(s(0))) =>* X:Nat .?

```
Maude> vu-narrow [1] minus(s(0),s(s(0))) =>* X:Nat .
vu-narrow [1] in SUM-LEQ : minus(1, 2) =>* X:Nat .

No solution.
```

Debido a que el s(s(0)) es 2 y el s(0) es 1, nos encontramos con un valor negativo al aplicar la resta, pero ese motivo no se devuelve solución ya que incumple la propiedad de que tienen que ser naturales.

2. ¿Qué devuelve la expresión vu-narrow [1] minus(s(s(0)),s(0)) =>* X:Nat .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow [1] minus(s(s(0)),s(0)) =>* X:Nat .
vu-narrow [1] in SUM-LEQ : minus(2, 1) =>* X:Nat .

Solution 1
rewrites: 2 in 0ms cpu (1ms real) (2002 rewrites/second)
state: 1
accumulated substitution:
variant unifier:
X:Nat --> 1
```

En este caso, se aprecia como s(s(0)) obtiene el valor 2, mientras que s(0) obtiene el valor 1, pudiéndose aplicar la resta satisfaciblemente y obteniendo un valor resultante de Nat 1.

3. ¿Qué devuelve la expresión vu-narrow [1] minus(X:Nat,s(s(0))) =>* Y:Nat .?

El resultado obtenido al realizar la ejecución de la expresión es la siguiente:

```
Maude> vu-narrow [1] minus(X:Nat,s(s(0))) =>* Y:Nat .
vu-narrow [1] in SUM-LEQ : minus(X:Nat, 2) =>* Y:Nat .

Solution 1
rewrites: 3 in 1ms cpu (2ms real) (1533 rewrites/second)
state: @1:Nat
accumulated substitution:
X:Nat --> s_^2(@1:Nat)
variant unifier:
Y:Nat --> %1:Nat
@1:Nat --> %1:Nat
```

En este caso, obtenemos s_^2(@1:Nat) que significa lo mismo que s(s(@1:Nat) siendo @1:Nat una nueva variable generada por maude, por tanto eso significa que se producirá la resta entre 4 y 3, dando como resultado 1, por ese motivo Y tiene como resultado Y:Nat -> %1:Nat