FIUBA

75-62 Técnicas de Programación Concurrente II 2004

Mozart-oz

Ing. Osvaldo Clúa

Bibliografía: Mozart-oz puede conseguirse en http://www.mozart-oz.org/ y trae su documentación completa. Sin embargo, la forma mas rápida de entenderlo es leyendo el Apéndice B de *Concepts, Techniques and Models of Computing Programs* de *Seif* y *Van Roy*. Este libro se usa demás en *Teoría de Lenguajes* que tiene una página no-oficial en *http://www.lfcrarezas.com.ar/mat7529/mat7529.htm*

Introducción

Oz es un lenguaje de programación multi-paradigma (puede usarse en forma procedural, funcional, con restricciones lógicas u orientado a objetos) que soporta programación en soft-real time, concurrencia, distribución y programación reactiva.

Mozart es el sistema de computación que soporta a oz y le permite la comunicación con el resto del mundo (se encarga de los *sockets*, I/O, interface gráfica, etc). Se desarrolló por investigadores de DFKI (German Research Center for Artificial Intelligence), SICS (Swedish Institute of Computer Science), la Universidad de Saarland, UCL (Université catholique de Louvain) y otros.

Oz está definido en función de un lenguaje *Kernel* y el resto de las construcciones de éste lenguaje se pueden considerar como *azúcar sintáctica* de este *Kernel*.

La BNF del lenguaje Kernel es:

<Statement> ::=<Statement1><Statement2>
|X = f(l1:Y1...ln:Yn)
|X = <number>
|X = <atom>
|X = <boolean>
|{NewName X}
|X= Y
|local X1 ... Xn in S1 end
|proc {X Y1 ... Yn}S1 end
|{X Y1 ... Yn}
|{NewCell Y X}
|Y =@ X
|X := Y
|{Exchange X Y Z}
|if B then S1 else S2 end
|thread S1 end
|try S1 catch X then S2 end
|raise X end

Las ***variables*** de *oz* son *lógicas*. Una vez que se ligan (*bind)* a un valor, éste valor no puede cambiarse (lo que se conocen como *variables* en otros paradigmas y que son entidades que mantienen estado pueden lograrse usando ***cells***). El ámbito de las variables es explícito de su declaración con *local.* En forma interactiva pueden introducirse con *declare.*

Los ***procedimientos*** se definen usando *proc* y se referencian luego por una variable. Las ***celdas*** se crean con *NewCell*, se leen con @ y se actualizan con := o con *Exchange*. Los **condicionales** usan la construcción *if* y las **excepciones** *raise* y *try.*

El ambiente de programación *Mozart* se presenta como una ventana *Emacs*. Cada división o ventana nueva se conoce como un *buffer* (notación *emacs*). *Emacs* como editor tiene un help que se invoca con *^h t* (*Control-h* y luego la tecla *t*, en formato emacs *c-h t*) En emcas, las combinaciones útiles son c-g (q*uit*, cancela el comando en proceso) y c-x u (undo) Se sale con c-x c-c.

Los comandos *emacs* de *oz* comienzan en general con c-. . Se escribe el código en el *buffer* llamado oz

Por ejemplo:

```
{Browse 'Hola Mundo'}

{Browse "Hola Mundo"}

{Browse 48*3}
```

Esta es una llamada al procedimiento *Browse* con un dato: 48*3 (La notación clásica de cálculo lambda). Se alimenta con esta línea al compilador (c-. c-l) y en el buffer "Oz compiler" se lee:

```
Mozart Compiler 1.3.0 (20040801) playing Oz 3


{Browse 48*3}
% -------------------- accepted
```

Además se abre una ventana con el *Browser* donde se despliega el resultado.

Un ejemplo algo mas completo es:

```
local Maximo A Be Ce in
   % primer ejemplo de oz
   proc {Maximo X Y Z}
      if X>Y then Z=X  else Z=Y  end
   end
      A=4
      Be=5
      {Maximo A Be Ce}
      {Browse Ce}
end
```

Las variables y los nombres de procedimiento deben comenzar con mayúsculas y

declararse. Los comentarios comienzan con %.

Usando el lenguaje Kernel se logran las demás características de oz como:

- abreviaturas

```
if B1 then S1 elseif B2 then S2 else S3 end
```

- o módulos como

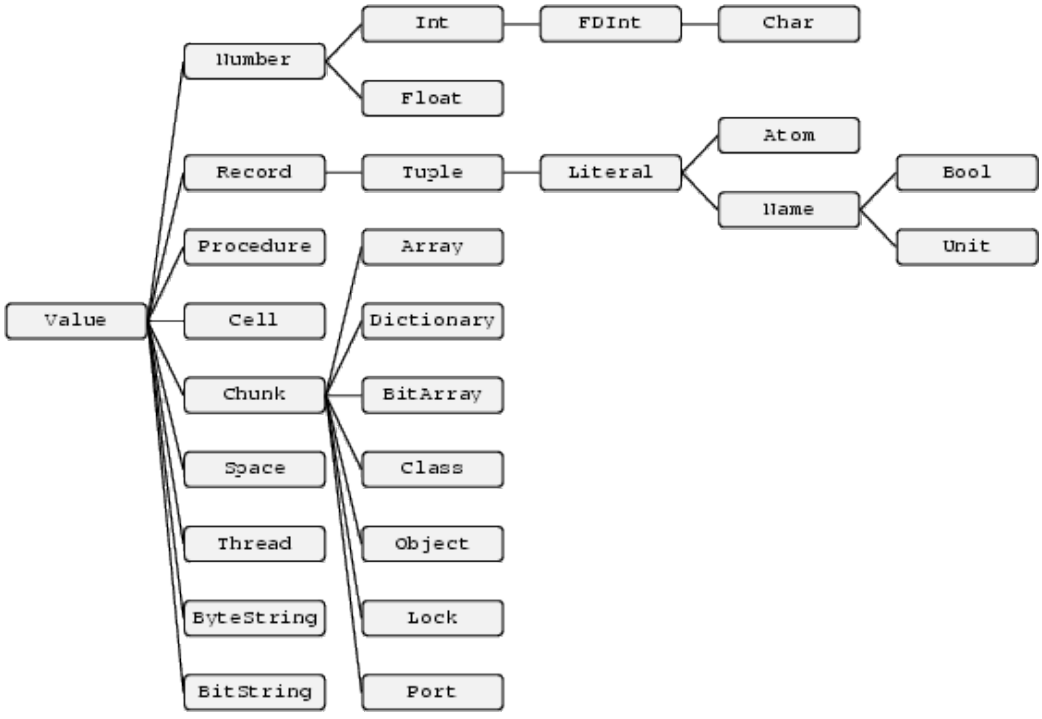
```
{For 1 11 3 Browse}
```

que dá como resultado

1 4 7 10

Los módulos se conocen además como *packages*.

La siguiente imagen, tomada de tutorial, muestra la estructura de los tipos primitivos de oz:



Los caracteres usan la codificación ISO 8859-1 (no Unicode) y su representación externa es numérica. Se introducen con & (por ejemplo, &t=116).

Los *records* son datos estructurados como en el siguiente ejemplo

```
local L L1 L2 V S in
   L=lista(val:5 sig:nil)
```

```
   L1=lista(V S)
   V=4
   S=nil
   L2=lista(3 L1)
   {Browse L2}
   {Browse L.val}
   end
```

L se define como una lista con 2 *features* (campos): **val** y **sig** (notar el uso de minúsculas). L1 se define como una lista en función de variables que aún no tienen valor (recordar que son variables lógicas, no celdas) EL segundo *Browse* muestra como seleccionar un campo. El procedimiento *Arity* indica la lista de las *features* tiene un record.

Las **lists** son las listas que implementa **oz** en forma nativa. Hay dos formas de escribir una lista:

- Una forma *cerrada* como L = [a 2 3 4]
- Una forma *abierta* según la definición que una lista es `nil` o un `valor` seguido del símbolo `|` seguido de una `lista` .

```
local L1 L2 in
L1=1|2|a|b
   {Browse L1}
   L2=[1 2 a b]
   {Browse L2}
end
```

La construcción abierta se puede usar en un procedimiento como en:

```
local L1 R1 L2 R2 Long in
   proc {Long L ?R}
      case L
       of nil then R=0
       [] _|Lr then local R1 in {Long Lr R1}
                    R=1+R1
                end
      end
   end
   L1=1|2|3|4|55|66|nil
   L2=[1 2 3 4]
   {Browse L1}
   {Long L1 R1}
   {Browse  R1}
   {Long L2 R2}
   {Browse R2}
end
```

Acá se usó la abreviatura *case*

```
case E of Pattern_1 then S1
[] Pattern_2 then S2
[] ...
else S end
```

Además se utilizó la seudovariable "_ " (subrayado) para denotar una variable que luego no será usada

El procedimiento Long es un dolor de cabeza, se podría haber usado una función que sería mas fácil de comprender.

```
local L1  L2 Long in
   fun {Long Ls}
      case Ls
      of nil then 0
      [] _|Lr then 1+{Long Lr}
      end
   end
   L1=1|2|3|4|55|66|nil
   L2=[1 2 3 4]
   {Browse  {Long L1} }
   {Browse  {Long L2} }
end
```

La mejor manera de probar estos códigos es insertarlos en la interface gráfica de Mozart (File-Insert o c-x i) o de abrirlos en un *buffer* usando los íconos de Emacs, copiar el código volver al *buffer* oz y pegarlo.

Se recomienda leer (aunque sea salteado) el Tutorial y el Ambiente Básico antes de continuar. Con estos elementos se puede atacar el nuevo paradigma de concurrencia. Las abstracciones de mayor nivel están en *módulos* y una buena enumeración de los disponibles está en el apéndice B del libro de Van Roi.

Finalmente, estos programas ejemplo se desarrollan con variables locales envueltos en `local ... in ... end` . Hay una forma mas interactiva de ir alimentando las líneas una a una al compilador introduciendo las variables con `declare ...`

Concurrencia Declarativa

Van Roy, Cap 4.

Un fragmento de código se dice que es *declarativo* si, siempre que se lo llame con los mismos argumentos devuelve los mismos resultados con independencia de algún otro estado. Es independiente, sin estado (nada se recuerda entre llamados) y determinístico.

Se demuestra los programas declarativos pueden componerse (probarse modularmente e integrarse) y que su estudio y comprensión se hace siguiendo reglas

algebraicas y lógicas simples.

Algunos programas se escriben mejor como un conjunto de actividades que se ejecutan en forma independiente interactuando solo cuando es necesario.

Las variables vistas hasta ahora, que pueden ligarse a un único valor, se conocen como declarativas. Desde el punto de vista de la concurrencia, estas variables pueden ya estar ligadas o ligarse en algún futuro. Por eso se las conoce como *dataflow variables.*

Como resultado de este tipo de variables, una *thread* puede ir calculando su resultado en la medida en que las variables que necesite se vayan ligando. La construcción de un thread se hace encerrando el código entre `thread ... end`

Veamos el siguiente fragmento:

```
declare X1 X2 X3
thread
   local  Y1 Y2 Y3 in
      {Browse [ Y1 Y2 Y3]}
      Y1 = X1+1
      Y2 = X2+Y1
      Y3 = X3+Y2
      {Browse completo}
   end
end
   {Browse X1 X2 X3}
```

Si se alimenta todo el párrafo a Oz, el *buffer* del *Browser* muestra `[ _ _ _ ]` indicando que la thread está a la espera de los valores y `[X1 _ _ ]` indicando que el principal tambien lo está.

Ahora, si alimentamos uno por uno las líneas

```
X1=1
X2=2
X3=10
```

Veremos como van apareciendo estos valores hasta que la thread se completa.

Este modelo es tan poderoso que elimina totalmente la *race condition* y el no-determinismo.

Al diseño basado en *dataflow* se lo conoce como *dataflow computation.*

Dataflow Computation

Primero el viejo conocido **productor-consumidor:**

```
declare Genera Sum in
fun {Genera N Lim}
   if N<Lim then N|{Genera N+1 Lim}
   else nil
   end
end
fun {Sum Xs A}
   case Xs of
      X|Xr then {Sum Xr A+X}
```

```
      [] nil then A
      end
   end
```

El procedimiento *Genera* crea una lista y *Sum* la acumula. Ahora la ejecución concurrente:

```
   local Xs S in
      thread Xs={Genera 0 100} end
      thread S={Sum Xs 0} end
      {Browse S}
      end
```

También así pueden generarse consumidores múltiples:

```
   declare Cont
   fun {Cont Xs A}
      case Xs of
         X|Xr then {Cont Xr A+1}
      [] nil then A
      end
   end

   local Xs S1 S2 S3 in
      thread Xs={Genera 0 100} end
      thread S1={Sum Xs 0} end
      thread S2={Cont Xs 0} end
      thread S3={Cont Xs 100} end
      {Browse [S1 S2 S3]}
   end
```

En realidad, los consumidores no "consumen" la entrada, sino que la leen. A esta construcción le podemos agregar un filtro:

```
   declare Impar in
   fun {Impar X} X mod 2 \= 0 end

   local Xs Ys S in
      thread Xs= {Genera 0 100} end
      thread Ys= {Filter Xs Impar} end
      thread S= {Cont Ys 0} end
      {Browse S}
      end
```

La función *Filter* es standard (y obvia). Los filtros son threads que pueden crearse dinámicamente como en la criba de Eratóstenes:

```
   declare Criba in
   fun {Criba Xs}
```

```
      case Xs
      of nil then nil
      [] X|Xr then
         Ys in
         thread
           Ys={Filter Xs fun {$ Y} Y mod X \=0 end}
         end
         X|{Criba Ys}
      end
   end
```

El $ se usa para no ponerle nombre a la *func* (uso parecido al _). La función toma cada valor de X y genera un thread que elimina sus múltiplos. Su invocación:

```
   local Xs Ys in
      thread Xs= {Genera 2 200} end
      thread Ys={Criba Xs} end
      {Browse Ys}
   end
```

Si el diseño impone *Barriers* se puede usar *Wait.*

Lazy Evaluation

Hasta ahora, el productor produce a su velocidad. Esta técnica se llama *eager computation.* Existe un procedimiento *ByNeed* que recibe una función sin parámetros y que sólo ejecuta la función cuando se precisa del valor que la produce. Por ejemplo:

```
   declare  X in
   X={ByNeed fun {$} {Browse 'invocado'} 4 end}
```

El resultado del *Browse* y la ejecución se darán cuando se precise el valor de X, por ejemplo al ingresar:

```
   {Browse X+1}
```

Oz tiene una abreviatura para todo esto: el atributo *Lazy:*

```
   declare X Cuatro in
   fun lazy {Cuatro} {Browse 'invocado'} 4 end
```

Las funciones *Lazy* no se ejecutan al llamarse, tampoco se bloquean, sino que esperan a que se precise el valor que calculan. Todas las funciones *Lazy* usan un thread sin que haga falta ponerlo en forma explícita. El productor anterior se puede programar entonces:

```
   declare LGenera  in
   fun lazy {LGenera N }
       N|{LGenera N+1}
      end
```

y usarse la misma invocación concurrente. Lo que cambia ahora es que los

números se generan cuando se necesiten (y no tiene sentido ponerle un límite a esa generación). Donde sí hay que ponerlo es en el consumidor:

```
declare LSum
   fun {LSum Xs A Lim}
      if Lim > 0 then
  case Xs of
     X|Xr then {LSum Xr A+X Lim-1}
  end
      else A end
   end
```

Esta técnica puede aplicarse también a los filtros.

Ejercicios

1) El procedimiento base de *List **Map*** recibe dos parámetros: una *list* y una *fun* y aplica la *fun* a cada elemento de la *list.* Por ejemplo:

```
declare Cuad Xs Xs1 in
fun {Cuad X} {Browse X} X*X end
Xs1=[120 2 30]
Xs={Map Xs1 Cuad}
{Browse Xs}
```

Construya una versión concurrente de Map.

2) *FoldL* "reduce" una list a partir de un valor inicial con un procedimiento P de dos parámetros como en

```
declare Max Xs Xr in
fun {Max X Y}
   if X>Y then X else Y
   end
end
Xs=[25 4 56 7]
Xr={FoldL Xs Max 0}
{Browse Xr}
```

Construya una versión concurrente (revise "The Oz Base Environment" de la documentación para ayudarse con las funciones de listas. Recuerde el cálculo de prefijos.

3) Simule el circuito de un sumador de 4 bits con carry usando filtros (versión Lazy).

4) Construya un decodificador para un código Huffmann usando filtros.