Oz Mozart

Daniel Delgado, *Estudiante, ITCR,* Wilbert Gonzales, *Estudiante, ITCR,*
Anthony Leandro, *Estudiante, ITCR,* and Bryan Mena, *Estudiante, ITCR*

1. Datos Historicos

Concebido en 1991 por Gert Smolka en la universidad de Saarland y desarrollado en colaboración con Seif Haridi y Perter Van Roy en el SICS (Swedish Institute of Computer Science), desde 2005 recibe mantenimiento por *Mozart Board*. Tiene herencia de:

Figura 1. Logo de Mozart

- Prolog
- Earlang
- LISP/Scheme

Oz es un lenguaje multiparadigma. Incluye las siguientes características que lo hacen un lenguaje muy interesante para enseñar e investigar:[1]

- Imperativo(Stateful) y funcional(Stateless)
- Data-Driven y Demand Driven programming
- Programación Relacional (Lógica) y Constraint-Propagation
- Concurrent and distributed programming
- Orientación a Objetos

1.1. Mozart

Mozart es un implementación de OZ, es un lenguaje de alto nivel desarrollado por la Université Catholique de Louvain para propositos educativos.

2. Tipos de Datos[2]

En Oz las variables no son variables, en OZ las variables son identificadores y no estan asignadas sino unificadas

- Dynamically Type Language
- Cuando una variable es creada su tipo y su valor son desconocidos
- Solamente cuando una variables asociada con un valor se determina su tipo

1. Datos tomados de *Programming in Oz*, Kuśnierczyk, W.
2. Por comodidad, adjuntamos imagen con tipos de datos en el Apéndice A. Imagen tomada de *Tutorial of Oz*

2.1. Estructuras Básicas de datos

- Numbers:
  Float: Es necesario que tengan decimales, en OZ 5.0 != 5
  Integer: OZ soporta formato binario, octal y hexadecimal para su representación
- Record: Compuesto de una etiqueta y un numero fijo de elementos
  Open Records: Igual que un Record pero con número variable de elementos
  Agrupar datos
  Ejemplo de Record:

Cuadro 1. Record

```
<Etiqueta>(Feature:Field)
```

Donde etiqueta es el nombre asociado al record, Feature es una etiqueta del elemento y field es el elemento, el conjunto de todas las etiquetas de Features se les llama arities (Algo similar a las keys en un diccionario de Python)

Se utiliza la notación ”.”para accesar a un elemento de un record. Ejemplo:

Cuadro 2. Accesar al elemento de un Record

```
{Browse <Etiqueta>.Feature}
```

Donde etiqueta es el nombre del record y Feature la etiqueta del elemento que se quiere accesar

- Literals: Tipos donde sus miembros no tienen estructura interna
  *Atoms: Records vacios, solo que únicamente contine una etiqueta y no tiene ninguna característica
  *Names: Es un identificador universal y único, la unicamanera de crearlo es llamando a `{NewName <Etiqueta>}`, su uso es importante ya que ayuda a la seguridad, como se ve en el arbol de tipos de datos un hijo de Name es Bool, esto hace que los valores true y false sean un Name por si solo, o sea únicos, universales e invariantes
- Tuplas
  Es un tipo de record, consiste de una etiqueta y valores
  Ejemplo

Cuadro 3. Tupla

```
<Etiqueta>(Feature:Field)
```

En realidad las tuplas son records donde los features son numeros desde 1 hasta la cantidad de elementos de la tupla:

Cuadro 4. Tupla

```
<Etiqueta>(1:elemento1 2:elemento2)
```

- Listas: puede ser el atomo nil para representar una lista vacia o puede representar una tupla usando el operador infijo | y dos argumentos, los cuales son la cabeza y la cola de la lista, otra representación utili es la lista cerrada representada por [ ] y separando elementos por espacios. Tambien estan las listas representadas con los caracteres " (doble comilla al inicio y al final) estos son los string, se puede utilizar el estatuto ForAll para iterar sobre cada elemento que posee

Algunas ideas sobre los tipos de datos anteriores:

- **Chunks**
  Prmite al usuario introducir tipos de datos abstractos
- **Cell**
  Modificar el estado de la lógica
- **Space**
  Resolución de problemas utilizando *"Search Techniques"*

3. Estructuras de Control y Expresiones

```
<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
```

Figura 2. The Oz kernel language

3.1. Operadores Condicionales

Operador	Significado
==	Igualdad
>	Mayor que
<	Menor que
>=	Mayor Igual que
<=	Menor Igual que
\=	Diferente

3.2. Operadores Booleanos

Operador	Significado
$false$	Valor de falsedad
$true$	Valor de Verdad
Not	Negación Lógica
Or / And	Or Lógico* And Lógico
$orelse$ / $andthen$	Short circuit de los anteriores

3.3. Declaración de variables

Para la declaración de variables que pertenecen a un scope dado se utiliza

Cuadro 5. Variables en un scope

```
local X Y Z in S end
```

El código anterior crear 3 variables (X, Y, Z) y ejecuta S. Usualmente las variables inician con mayúscula seguido de cualquier cantidad de caracteres alfa numericos. Otra manera de declarar variavles es:

Cuadro 6. Variables en un scope

```
declare X Y Z in S
```

Lo que esto hace es que X, Y, Z sean visibles globalmente en S y en los estatutos que sigan a S

En Oz hay pocas maneras de asociar variables a un valor, la usual es utilizar el operador infijo " = ", ahora bien, si una variable ya contiene un valor la operación es considerada un test.

3.3.1. Qué pasa si se hace X = Y?

Cuando se crea una variable se le asigna un espacio en memoria, un nodo, este nodo al inicio tiene valor y tipo desconocido, cuando las referencias a esa variable no existen se inicia un proceso de garbage collection para liberar los nodos que utilizaba esta varaible, cuando se utiliza la operación " = " intentará unificar los valores de X y Y copiando sus nodos. La operación " = " se conoce como *incremental tell* o *unification*, algunos de sus resultados dependiendo del contexto:

- Si las etiquetas X y Y pertenecen al mismo nodo la operación esta completa
- Si X no esta asociado se unifica el nodo de X con el nodo de Y, o sea todas las referencias a X pasan a ser referencias a Y
- Si X y Y contienen Records Rx y Ry respectivamente:
  Si los records Rx y Ry tienen diferentes etiquetas o arities se lanza un exception
  De otra manera los features de los records son unificados

3.4. Operador de Igualdad

Para probar una igualdad se utiliza l código:

Cuadro 7. Variables en un scope

```
{Value.'==' X Y R}
```

Lo que se hace es probar si X es igual a Y y dejar el resultado en R. La operación:

- Retorna true si los elementos tienen la misma estructura y los mismos valores o si son referencias al mismo nodo en memoria
- Retorna false si los elementos tienen estructuras o valores diferentes
- Se suspende cuando los nodos son diferentes pero existe un elemento sin asociar a un valor. Como Oz es un lenguaje concurrente cuando pasa esto el hilo que ejecutó el código se suspende tambien.

Tambien se puede utilizar el operador `"=="` como infijo tal que R = X == Y donde se prueba si X y Y son iguales y se deja el resultado en R

3.5. Estatutos IF

Cuadro 8. Variables en un scope

```
if B then S1 else S2 end
```

Lo que hace:

- Si B es true S1 se ejecuta
- Si B es false S2 se ejecuta
- Si B no es un valor Booleano una exception ocurre
- Si B no tiene un valor asociado el hilo ejecutando se suspende

Importante mencionar que la palabra reservada *skip* funciona como el *continue* en Python, además existe la abreviación *elseif* que vendría siendo algo similar al *elif* en Python:

Cuadro 9. Variables en un scope

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

3.6. Estatutos CASE

Cuadro 10. Variables en un scope

```
case E
of Pattern_1 then S1
[] Pattern_2 then S2
[] Pattern_3 then S3
[] ...
...
else S end
```

3.6.1. Semantica

Lo que hace el estatuto case es evaluar E con los $patrones_i$ esto de izquierda-derecha y *depth-first*. Lo que hace:

- Si E hace match con el $patron_i$ y E no esta siendo utilizado la instrucción S_i se ejecuta
- Si E hace match con el $patron_i$ pero E se esta utilizando se suspende el hilo
- Si E no hace match con el $patron_i$ se intenta con el $patron_i+1$ asi hasta alcanzar el else que se ejecutaria por defecto
- Dado sea el caso que la parte del *else* sea omitida si E no hace match con algun $patron_i$ se lanza un exception

3.7. Loops

3.7.1. For

El procedimiento For From To Step P es una abstracción de un for qu aplicac el prodecimiento P, From y To son integers que denotan el inicio y el fin del ciclo, Step es otra integer que indica el incremento (o decremento) en From para llegar a To. Un ejemplo de For:

Cuadro 11. For

```
{For 1 10 1 Browse}
```

El código anterior mostrará los números de 1 al 10. Ejemplo más general

Cuadro 12. For

```
local
   proc {HelpPlus C To Step P}
      if C=<To then
         {P C} {HelpPlus C+Step To Step P}
      end
   end
   proc {HelpMinus C To Step P}
      if C>=To then
         {P C} {HelpMinus C+Step To Step P}
      end
   end
   in proc {For From To Step P}
      if Step>0 then
         {HelpPlus From To Step P}
      else
         {HelpMinus From To Step P}
      end
   end
end
```

3.8. Manejo de Excepciones

Para lanzar una excepción se utiliza

Cuadro 13. Variables en un scope

```
raise E end
```

donde E es una expresión de error Usualmente se utiliza un try-statement para manejar estos errores:

Cuadro 14. Variables en un scope

```
try S1 catch X then S2 end
```

La anterior es la forma más simplificada de un try-statement:

Cuadro 15. Variables en un scope

```
try S catch
Pattern_1 then S1
[] Pattern_2 then S2
...
[] Pattern_n then Sn
finally
S_final
end
```

Lo anterior se puede utilizar asemejando el try catch de Java con varios bloques catch y el bloque finally

3.9. Procedimientos

Cuadro 16. Variables en un scope

```
proc {P X1 ... Xn} S end
```

El código anterior lo que hace es crear una lambda expression única lo que lo hace diferente a todos los demás procedimientos existentes, esa expresión esta asociada con el valor P. Como dato curioso, la equicvalencia de procedimientos se realiza mediante su etiqueta o nombre. Ejemplo:

Cuadro 17. Obtener el mayor entre dos números

```
local Max X Y Z in
        proc {Max X Y Z}
                if X >= Y then Z = X
                else Z = Y end
        end
        X = 5
        Y = 10
        {Max X Y Z} {Browse Z}
end
```

3.10. Funciones

Cuadro 18. Variables en un scope

```
fun {F X1 ... Xn} S E end
```

Como se puede apreciar la sintaxis de una función es muy similar a la de un procedimiento. Oz permite ciertas formas de optimización tail-recursion

3.11. Funciones y procedimientos Anónimos

La idea detras de un procedimiento anónimo es que la función o procedimiento no esta asociada con una etiqueta generalmente son usada para procedimientos o funciones que no son longevas. En mozart se utiliza el simbolo *$* en vez de pasar la etiqueta que contendria la funcion un ejemplo de esto

Cuadro 19. Variables en un scope

```
fun {$ X1 ... Xn} ...end
```

La función anterior no estaria asociada con una etiqueta simplemente existe. Unejemplo útil de esto:[3]

Cuadro 20. Variables en un scope

```
local
  Max = proc {$ X Y Z}
          if X >= Y then Z = X
          else Z = Y end
        end
  X = 5
  Y = 10
  Z
in
  {Max X Y Z} {Browse Z}
end
```

3. Tomado de *Tutorial of Oz*

4. Módulos e Interfaces

Comúnmente los modulos son un grupo de procedimientos, valores, entre otros que se encuentran contenidos en un mismo lugar con el fin de brindar un conjunto de servicios relacionados, usualmente estos modulos contienen procedimientos privados que solo son accesibles dentro de si

5. Orientación a Objetos

En Oz una clase es un Chunk que contiene:

- Una colección de métodos
- Descripción de los atributos que cada instancia tendrá
- Descripción de los features que cada instancia de la clase tendrá (Un feature es una variable inmutable accesada por feature-name)
- Son simplemente descripciones de como un objeto se debe comportar

Ejemplo de una clase:

Cuadro 21. Variables en un scope

```
class Counter
  attr val
  meth browse
    {Browse @val}
  end
  meth inc(Value)
    val := @val + Value
  end
  meth init(Value)
    val := Value
  end
end
```

Algunos datos importantes:

- Para clases anónimas se utiliza el símbolo *$*
- La llamada a un metodo toma como argumento implícito el objeto actual (self)
- Existe una clase trivial (Base Object), el objetivo es que los hijos que la implementen no necesariamente tienen una función de inicialización
- es posible definir una clase abstracta donde los métodos se dejan sin especificación

Para definir una clase se utiliza la palabra clave *class*, para los atributos se definen con *attr* para los features(similares a los records) se utiliza *feat* y los métodos con *meth*; para herencia se utliza la palabra reservada *from*, para accesar a un atributo se utiliza @¡AttrName¿, para llamar a un método de una clase se utiliza ¡Class Name¿, ¡Method Name¿(..)

5.1. ¿Para que se utiliza *self*?

Como en muchos lenguajes de Orientación a objetos, self se utiliza en vez de clases especificas, algo as como un binding dinámico, donde *self* tomaá el valor de la clase desde la cual se invoca algún elemento

5.2. Atributos

Como ya se dijo anteriormente, los atributos se declaran utilizando la palabra reservada *attr*.

- Los atributos son privados y solo los puede manipular el objetos que los contiene, la única manera de modificar un atributo fuera de la clase es que la misma clase tenga métodos para hacerlo
- Son *Cells* que se pueden asignar, reasignar y accesar a voluntad

5.3. Argumentos por Default para un Método

Algunas veces en llamada a métodos puede suceder que se desee no brindar un parametro, para esto existen los valores por default. Lo que se hace es asignar al argumento que no recibió un valor un valor por default, esto se hace con el siguiente coódigo:

Cuadro 22. Variables en un scope

```
meth m(X Y d1:Z<=0 d2:W<=0) ... end
```

En caso que no se un valor para d1 o d2 se asume el valor de 0

5.4. Herencia

En Oz la herencia multiple esta implementada, se considera una clase (B) superclase de otra (A) si:

- La clase B aparece despues de la declaración *from* de la clase A
- La clase B es superclase de una clase C que aparece en la declaracion *from* de la clase A

La herencia es una herramienta para construir clase apartir de clases ya existentes dandole un rango de metodos y atributos que tendrá la nueva clase. Un método de la clase A sobre escribe cualquier método con la misma etiqueta que posean sus superclases. No se permiten herencias ciclicas, esto es:

Cuadro 23. Variables en un scope

```
class A from B ... end
class B from A ... end
```

No se permite que en herencia dos superclases (o más) tengan métodos atributos o features con el mismo tag esto no se evalua en tiempo de compilacin sino en tiempo de ejecución cuando se crea una instancia del objeto y se intenta acceder al atributo o método con el tag repetido

6. Características

- Compilado o Interpretado, implementado en la plataforma Mozart
- Seguro, las entidades son creadas y pasadas explicitamente, esto significa que una aplicación no puede accesar o dar acceso a referencias que no se le han dado o creado en si misma
- Multiparadigma
  - Orientación a Objetos
  - Programación Lógica
  - Programación Concurrente
    - Threads Dinámicos

7. Ventajas

- Lenguaje Multiparadigma
- Concurrencia, hilos de pesos ultraligeros
- Lenguaje flexible

8. Desventajas

- Debido a la flexibilidad es un poco lento, se han realizado pruebas donde OZ es 50 % más lento que un compilador de C

Referencias

[1] Oz Mozart Home Page, http://mozart.github.io/
[2] *Mozart Programming System*, P. Alarcon, H. Spakes, J. Ward; Arkansas Tech University
[3] *Tutorial of Oz*, S. Haridi, N. Franzn, Recuperado de: http://mozart.github.io/mozart-v1/doc-1.4.0/tutorial/index.html
[4] *A review of Oz and its implementation with Mozart*, Philippe Giabbanelli, Bishops University, Recuperado de: http://aqualonne.free.fr/Teaching/csc/oz.pdf

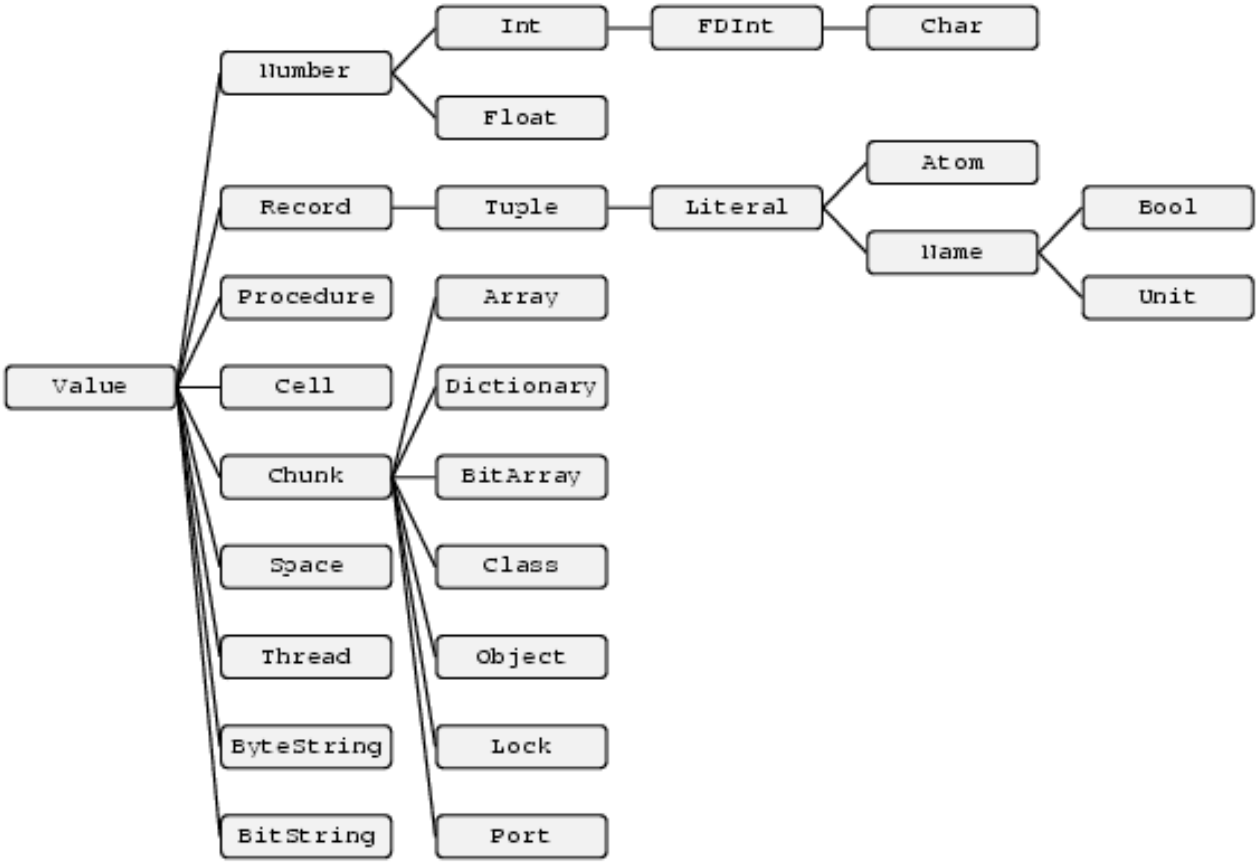

Figura 3. Tipos de Datos, Tomado de *Tutorial of Oz*

APÉNDICE A