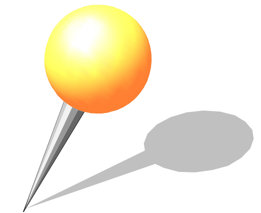


Oz: un moderno linguaggio di programmazione multi-paradigma

Federico Scarel

Quest'opera è stata rilasciata sotto la licenza
Creative Commons Attribution-NonCommercial-ShareAlike 2.0 Italy.
Per leggere una copia della licenza visita il sito web http://creativecommons.org/licenses/by-nc-sa/2.0/it/
o spedisci una lettera a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Cos'è Oz ?

- Multiparadigma:
  - Dichiarativo
  - Imperativo
  - Ad oggetti
  - Constraint programming
  - ...
- Il codice di Oz viene compilato per essere eseguito su una macchina astratta (come Java, C#, ...)
- Garbage collector
- Non è staticamente tipizzato (es. C, Pascal), ma ha una tipizzazione dinamica

Cos'è Mozart ?

- Ambiente di sviluppo Open Source per il linguaggio Oz
- Contiene anche la vm di Oz
- Disponibile per diversi sistemi operativi: Linux, Windows, OS X, diversi altri *nix
- IDE basato su Emacs
- Facilita lo sviluppo incrementale del codice
- Offre diversi strumenti per lo sviluppo del codice:
  - Oz panel
  - Distribution panel
  - Debugger
  - Oz Explorer

Ambiente di sviluppo

Oz Programming Interface (emacs@cucciocoso.randaccio)

File Edit Options Buffers Tools Oz Help

```
declare
Xs
Ys

fun {Genera Start End}
   if Start < End then
      Start | {Genera Start+1 End}
   else nil end
end


fun {Filtro Xin}
   case Xin
   of nil then nil
   [] X | Y then Ys in
      thread
         Ys = {Filter Y fun {$ A} A mod X \= 0 end}
      end
      X | {Filtro Ys}
   end
end
-u:** Oz          (Oz)--L6--C14--Top---------------------------------------
end


   thread Xs = {Genera 2 100000} end
   thread Ys = {Filtro Xs} end
   {Browse Ys}
Declared variables:
  Filtro: procedure/2
  Genera: procedure/3
  Xs: value
  Ys: value
% -------------------- accepted

-u:** *Oz Compiler*    (Compilation:open)--L150--C0--Bot-----------------------
menu-bar Oz Feed Buffer
```

Oz Panel

Panel Options

Threads | Memory | Problem Solving

Runtime

Run:	11.29	s
Garbage Collection:	3.17	s
Copy:	0.00	s
Propagation:	0.00	s

Threads

Created:	10393
Runnable:	4

```
9|23|29|31|37|41|43|47|53|59|61|67|
101|103|107|109|113|127|131|137|
3|167|173|179|181|191|193|197|199|
3|239|241|251|257|263|269|271|277|
1|313|317|331|337|347|349|353|359|
9|397|401|409|419|421|431|433|439|
3|467|479|487|491|499|503|509|521|
3|569|571|577|587|593|599|601|607|
1|643|647|653|659|661|673|677|683|
7|733|739|743|751|757|761|769|773|
1|823|827|829|839|853|857|859|863|
7|911|919|929|937|941|947|953|967|
7|1009|1013|1019|1021|1031|1033|
1|1063|1069|1087|1091|1093|1097|
3|1129|1151|1153|1163|1171|1181|
3|1217|1223|1229|1231|1237|1249|
3|1289|1291|1297|1301|1303|1307|
1|1367|1373|1381|1399|1409|1423|
9|1447|1451|1453|1459|1471|1481|
3|1499|1511|1523|1531|1543|1549|
```

LinuxDay 2005 - Ko | Oz Programming Int | Gimp [2] | presentazione_oz.s | OzTools [2] | 12:22

Tool: debugger

Oz Debugger

Ozcar Action Thread Stack Options Help

SubThreads: Attach Queries: Attach

Thread Forest

```
3232
 ├ 3369 *
 ├ 3447
 ├ 3528
 └ 3612
    ├ 4024
    ├ 4174
    ├ 4327
    ├ 4483
    └ 4642
```

Stack of Thread 3369

```
-> 1 loop
-> 2 conditional <tuple>
<- 3 {Number.'*' 5 5 25}
```

Local Variables

```
Msg        <tuple>
X          5
Y          5
R          25
```

Global Variables

```
<N: ForProc>  <proc>
Port          <port>
Stream        <list>
```

Tool: explorer

Oz Explorer

Explorer Move Search Nodes Hide Options

Time: 110ms 499 9 481 Depth: 17

Tool: panel

Oz Panel

Panel Options

Threads | Memory | Problem Solving

Runtime

Run:	0.39	s	
Garbage Collection:	0.08	s	
Copy:	0.03	s	
Propagation:	0.00	s	

1.0
0.8
0.6
0.4
0.2
0.0

Threads

Created:	18329
Runnable:	1

5
4
3
2
1
0

Tipi di dato [1]

- Value
  - Number
    - Int
      - Char
    - Float
  - Record
    - Tuple
      - List
        - String
      - Literal
        - Bool
        - Atom
      - ...
  - Cell
  - Chunk
    - Array
    - Dictionary
    - Lock
    - Port
    - ...
  - Procedure
  - Space
  - ...

Tipi di dato [2]

- Oz è un linguaggio di programmazione con tipizzazione dinamica
- Possiede sia variabili a singolo assegnamento che le normali variabili dotate di stato (celle di memoria)
- Usa un sistema di garbage collection per deallocare le variabili non più utilizzate

Le variabili di Oz

- Sono a singolo assegnamento. Possono essere unbounded oppure assegnate a qualche valore
- Variabili più complesse (record, liste), possono essere vincolate parzialmente.
- Viene usato l'algoritmo di unificazione per vincolare le variabili

Esempi

Codice	Risultato
`X = 1` `X = 2`	Failure
`X = Y` `Y = 5`	X = 5

`X = f(W Z)`
`Y = q(W Z)`
`if (X \= Y) then`
`...`
	Viene eseguito il ramo then

`X = f(a Z)`
`Y = f(Z a)`
`Z = a`
	X e Y risultano uguali

Oz come linguaggio imperativo

- Sono presenti variabili dotate di stato
- X = {NewCell 0}: crea una nuova variabile di tipo cell con valore inziale 0
- Uso delle celle:

X := @Y + @X

Assegnamento

Valore di Y. È l'equivalente di {Cell.access Y}

Gestione delle eccezioni

- raise → solleva un'eccezione
- Gestione delle eccezioni sollevate da raise tramite il classico

```
try
    codice_da_eseguire
catch x
    case x
    of eccezione_1 then codice_gestione_eccezione_1
    [] eccezione_2 then codice_gestione_eccezione_2
    ...
    [] eccezione_n then codice_gestione_eccezione_n
end
```

OOP e Oz: classi

```
class File
    attr name
    feat mode
         OS: linux
    meth write(In)
       ...
    end
    meth read(?Out)
       ...
    end
    ....
end
```

- Attributi: sono variabili private, dotate di stato, che ogni oggetto possiede.
- Metodi: procedure che operano sugli attributi degli oggetti
- Features: sono variabili a singolo assegnamento. Se viene fatta una qualche forma di binding nella dichiarazione della classe, sono “per classe”, altrimenti sono “per oggetto”

Ereditarietà

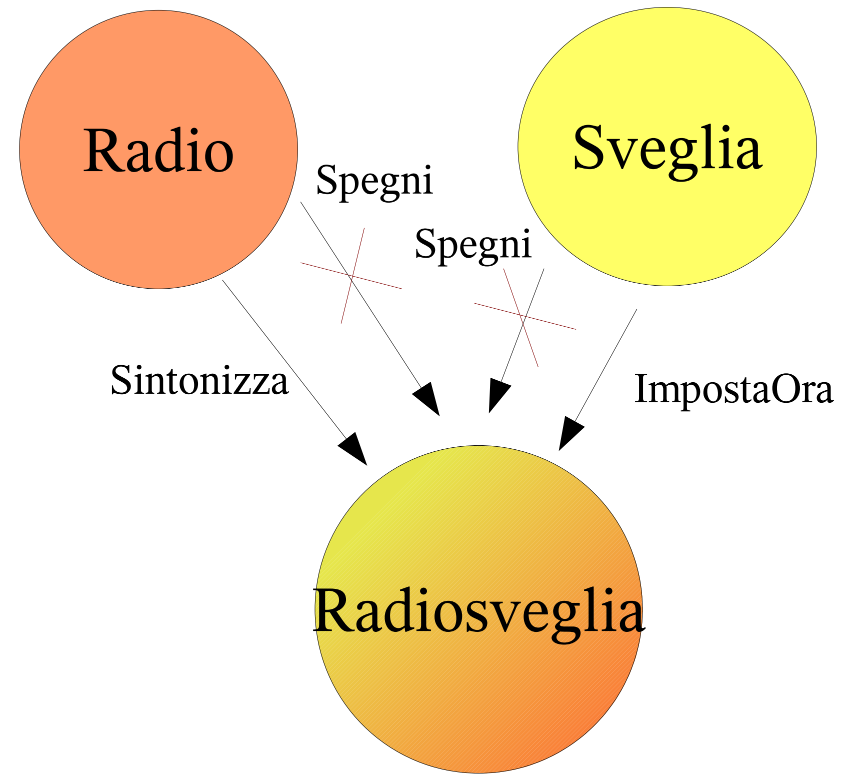
- Singola:

```
class Base
    meth faiQualcosa(Input)
    ...
    end
    ...
end
class Derivata from Base
    meth faiQualcosa(Input)
        Base,faiQualcosa(Input)
        {self FaiAltro(Input)}
    end
    ...
end
```

Ereditarietà [2]

- Ereditarietà multipla

```
class Radio
    meth sintonizza(Freq)
    ...
    end
    ...
end
class Sveglia
    meth impostaSveglia(Ora)
    ...
    end
    ...
end
```




```
class RadioSveglia from Radio Sveglia
    meth snooze()
    ...
    end
    ...
end
```

Metodi pubblici e privati

- In Oz, gli attributi sono sempre considerati privati. L'accesso ai dati viene quindi sempre mediato dai metodi
- **Accesso pubblico:** la dichiarazione del metodo inizia con una minuscola
- **Accesso privato:** la dichiarazione del metodo inizia con una maiuscola
- Tale distinzione, dal punto di vista del linguaggio, ha un perché:
  - I metodi dichiarati con la maiuscola vengono tradotti in Name **locali** alla classe
  - I metodi dichiarati con la minuscola sono invece tradotti in atomi (dei letterali)

Metodo otherwise

- Per intercettare chiamate a metodi non dichiarati, si può implementare il metodo otherwise

```
class Square
    ...
    meth area(A)
        A = @l * @l
    end
    meth otherwise(M)
        {Browse 'Metodo '#M#' non implementato'}
    end
end


Q = {New Square init}
{Q raggio(R)}
```

Concorrenza dichiarativa

- Creazione di thread estremamente semplice: thread ... end
- Gestione dei thread particolarmente efficiente: si possono creare tranquillamente migliaia di thread
- Usando variabili a singolo assegnamento, la sincronizzazione è implicita
- Problema: l'ordine (parziale) di computazione viene determinato dal binding delle variabili. Anche per questo tali variabili vengono chiamate “Dataflow”

Esempio di codice

```
thread
    Y = X + 1
    Z = X + 2
end
thread
    {Browse Y}
end
thread
    {Browse Z}
end
X = 6
K = Z + Y
```

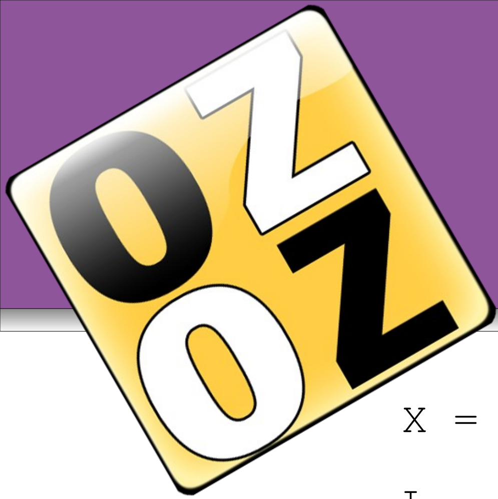
Possibile ordine di esecuzione

Concorrenza con stato

- La concorrenza con variabili dataflow è poco adatta a numerose tipologie di problemi:
  - Uso di celle
  - Gestione di eventi non “prevedibili” (es. eccezioni, gestione di eventi “esterni” asincroni, ...)
- Oz fornisce una serie di strumenti “classici” per gestire queste tipologie di problemi

Lock rientranti

- I lock (rientranti) consentono di bloccare l'accesso a porzioni di codice (**anche non contigue**) ad un solo thread alla volta
- Il primo thread che esegue quel codice ne richiede il “locking” ed “entra” nella sezione di codice “lockata”. Nessun altro thread può entrare in tale sezione (rimangono sospesi sulla richiesta di lock) fino a che il primo thread non esce dal codice “lockato”
- Una volta uscito, uno degli altri thread che richiedevano il locking della sezione può entrarvi



Esempio codice: lock

```
X = {NewCell 0}
L = {NewLock}
proc {Inc}
    lock L then X := @X + 1 end
end
proc {Dec}
    lock L then X := @X - 1 end
end
for I in 1..100 do
    if ((I mod 2) == 0) then
        thread {Inc} end
    else
        thread {Dec} end
    end
end
```

Lock e classi

```
class ConcurrentCounter
   prop locking    %% locking implicito
   attr count
   meth inc(Value)
      lock
         count := @count +1
      end
   end
   ...
end
```

- L'effetto che si ottiene è simile al ***synchronized*** di Java

Canali di comunicazione: porte e socket

- Sono disponibili due tipi di canali di comunicazione: uno intra-processo (tra thread dello stesso processo)*, che è la port, ed uno inter-processo (tra processi distinti), cioè il socket
- Le port non coincidono con il concetto di port TCP, ma sono dei semplici canali di comunicazione FIFO asincroni concorrenti
- Alle socket viene invece assegnata la classica porta TCP, e possono quindi essere usate per comunicare all'interno di una rete basata su TCP/IP

*Questo non è più valido in ambito distribuito

Port

- Sono dei semplici canali di comunicazione (o code) FIFO molti a uno (più thread spediscono messaggi sullo stesso canale ed un thread li riceve)
- Vi sono principalmente due primitive per operare con i port:
  - *Port* = {**NewPort** *Stream*}, che crea un nuovo oggetto *Port* associato al canale *Stream*
  - {**Send** *Port Messaggio*}, che spedisce il messaggio *Messaggio* a *Port*

Esempio di codice: Port

```
Port = {NewPort Stream}

thread

    for Msg in Stream do

    case Msg

        of mangia then {Browse 'GNAM!'}

        [] dormi then {Browse 'ZZZZZ'}

        [] moltipica(X Y Result) then

            Result = X * Y

    end

end
```

```
thread
    for I in 1..10 do
        case {OS.rand} mod 3 + 1
        of 1 then {Send Port mangia}
        [] 2 then {Send Port dormi}
        [] 3 then local Result in
            {Send Port moltiplica(4 3
              Result)}
            {Browse Result}
           end
        end
    end
end
```

Socket [1]

- Struttura dati che consente la comunicazione tra processi dello stesso computer o di computer connessi da una rete
- Si possono creare i due classici tipi di socket:
  - ***Stream socket***, dove si apre un flusso di dati bidirezionale. I dati vengono ricevuti nello stesso ordine in cui sono stati spediti;
  - ***Datagram socket***, che sono essenzialmente connessioni con assenza di memoria e dove non vi è alcuna garanzia sulla ricezione.

Socket [2]

- Operazioni base:
  - MySocket = {New Open.socket init(tipe:stream)}
  - {MySocket bind(takePort:27)}
  - {MySocket listen}
  - {MySocket accept(host:HostCheHaInviato)}
  - {MySocket connect(host:google.com port:80)}
  - {MySocket receive(list:MessaggioLetto)}
  - {MySocket send(vs:"Hi!")}
  - {MySocket shutDown(how:[receive send])}
  - {MySocket close}

Esempio di codice: socket

```
HTTPServer = {New Open.socket init}

{HTTPServer bind(takePort: 80)}

{HTTPServer listen}

{HTTPServer accept}

{HTTPServer receive(list:Msg)}

{HTTPServer send(vs:”HTTP/1.1 200 OK\r\nServer:
   OzServer\r\nConnection: close\r\nContent-Type:
   text/html\r\n\r\n <html><body><h1>Hello
   World!</h1></body></html>”)}

{HTTPServer close}
```

Constraint Programming

- Il *constraint programming* (programmazione con vincoli) è un insieme di tecniche di programmazione per risolvere problemi di *constraint satisfaction* (soddisfacimento di vincoli)
- Un *constraint* (vincolo) non è nient'altro che una qualche forma di relazione logica, come ad esempio “X è minore di Y” oppure “Y è divisibile per 4”
- La programmazione con vincoli è dichiarativa: si è più interessati a *cosa* vogliamo risolvere piuttosto che *come* vogliamo risolvere un problema

Constraint Programming e Oz

- Oz consente di risolvere facilmente problemi di constraint programming
- Semplice problema di constraint satisfaction: trovare una sequenza di nove differenti cifre comprese tra 1 e 9 tali che:
  - [1] $C_4 - C_6 = C_7$
  - [2] $C_1 \times C_2 \times C_3 = C_8 + C_9$
  - [3] $C_2 + C_3 + C_6 < C_8$
  - [4] $C_9 < C_8$
  - [5] Il valore delle cifre non sia uguale al loro ordine nella sequenza (la prima cifra sia diversa da 1, la seconda da 2 ...)

Metodo di risoluzione classico

- Un (banale) metodo di risoluzione “classico” al precedente problema è quello “per tentativi ed errori” utilizzando il backtracking
- Si esplora l'albero dei possibili assegnamenti alle variabili finché non si incontra una soluzione valida
- Problema: la dimensione dell'albero da esplorare cresce esponenzialmente con il numero di variabili in gioco
- Occorre quindi pensare per ogni problema come eliminare dall'esplorazione il maggior numero possibile di rami che non portano ad una soluzione valida

Codice di esempio

```
proc {Esempio Sol}
    {FD.tuple ris 9 1#9 Sol} %%Sol è una tupla di interi tra 1 e 9
    {FD.distinct Sol}    %%Gli elementi di Sol sono tutti diversi
    Sol.4 - Sol.6 =: Sol.7
    Sol.1 * Sol.2 * Sol.3 =: Sol.8 + Sol.9
    Sol.2 + Sol.3 + Sol.6 <: Sol.8
    Sol.9 <: Sol.8
    for I in 1..9 do
        Sol.I \=: I
    end
    {FD.distribute ff Sol}   %%Faccio il distribute sulla tupla Sol
end
{ExploreAll Esempio}
```

Propagate & distribute

- In Oz viene usato un metodo chiamato "***propagate & distribute***" per cercare le soluzioni di un problema con vincoli
- Nella fase di propagazione, si cerca di ridurre i domini delle variabili tramite regole di inferenza
- Una volta che non si possono più usare delle regole per ridurre i domini, se non si è trovata una soluzione avviene la fase "*distribute*": vengono aggiunti nuovi vincoli compatibili con i domini delle variabili e si ripete la fase di propagazione

Esempio di propagate & distribute

- **Fase di propagazione:**

Dopo il "for": Sol.1 in {2..9}, Sol.2 in {1, 3..9}, Sol.3 in {1..2, 4..9} ...

Sol.2 + Sol.3 + Sol.6 < Sol.8 → Sol.2 in {1, 3..7}, Sol.3 in {1..2, 4..7}

Sol.4 – Sol.6 = Sol.7 → Sol.7 in {1..6, 8}

continuo fino a che non posso più applicare vincoli. Ottengo:

({2..9} {1, 3..6} {1..2,4..6} {2..3,5..9} {1..4,6..9} {1..5} {1..6,8} {4..7,9} {1..8})

- **Fase di distribute**: creo due nuovi spazi di computazione da eseguire concorrentemente. Ho scelto la politica "First Fail" e quindi scelgo il primo valore della seconda cifra, che è la prima variabile con dominio "più piccolo"

(a) Impongo il nuovo vincolo Sol.2 =: 1 sul primo spazio

(b) Impongo il vincolo Sol.2 \=: 1 sul secondo spazio

Ripeto quindi la fase di propagazione su entrambi gli spazi. Alla fine giungo a spazi non soddisfacibili (es. X in {1..3} e X > 4) oppure a spazi risolti.

Oz in ambiente distribuito

- E' possibile condividere in modo trasparente variabili, celle, funzioni, port, etc tra processi, anche se i processi risiedono su più computer
- Il garbage collector funziona anche in ambito distribuito
- Vi sono quattro moduli che consentono di creare programmi distribuiti in Mozart:
  - Connection, che offre la possibilità di condividere entità
  - Pickle, che consente di salvare/recuperare strutture stateless su file o URL
  - Remote, che consente di creare processi su macchine host tramite rsh o ssh
  - Fault, per il rilevamento e la gestione di errori

Condividere variabili

- MyTicket = {Connection.offer LocalEntity}: crea un ticket da poter offrire per la condivisione di LocalEntity. Il ticket può essere preso solo una volta.
- MyTicket = {Connection.offerUnlimited LocalEntity}: come sopra ma il ticket può essere preso un numero illimitato di volte
- {Connection.take MyTicket Entity}: assegna ad Entity il riferimento contenuto nel ticket MyTicket. Dopo l'esecuzione di questa operazione, Entity risulterà dal punto di vista del linguaggio la stessa entità a cui MyTicket fa riferimento. Se ad esempio Entity fa riferimento alla variabile X di un processo su un altro computer, fare X=1 farà diventare Entity=1 e viceversa.

Salvare strutture stateless

- {Pickle.save MyStruct “public_html/nomefile”}: consente di salvare l'entità stateless MyStruct nel file *public_html/nomefile.* Sono quindi salvabili classi (non oggetti), procedure, atom, funzioni, etc ma non variabili, celle, oggetti o altre entità dotate di stato.
- MyOtherStruct = {Pickle.load “http://www.sito.org/~gigi/nomefile”}: preleva la struttura dall'URL fornito e la assegna a MyOtherStruct.

Esempio di codice distribuito

- Programma A

```
declare Result
fun {Fattoriale N}
    if N > 1 then
        N * {Fattoriale N-1}
    else
        1
    end
end
TicketFun = {Connection.offer Fattoriale}
TicketVar = {Connection.offer Result}
{Pickle.save TicketFun “fun_file”}
{Pickle.save TicketVar “var_file”}
{Browse Result}
```

- Programma B

```
MyFun = {Connection.take {Pickle.load
   “fun_file”}
MyVar = {Connection.take {Pickle.load
   “var_file”}

MyVar = {MyFun 5}
```

Alcuni riferimenti

- http://www.mozart-oz.org [sito ufficiale del consorzio Mozart]
- P. Van Roy, S. Haridi: Concepts, Techniques, and Models of Computer Programming [libro sulla programmazione che usa Oz come linguaggio di riferimento]