Programming in Oz

Wacek Kuśnierczyk
December 10., 2010

Lecture Outline

Introduction to Oz
- Oz & Mozart
- Playing Oz

Programming in Oz: Basics
- `mdc`, the Little Brother

Programming Oz: More Features
- Last-Call Optimization and Tail-Recursion
- Dataflow Variables
- Concurrency, Streams, Synchronization
- Lazy Evaluation
- Message Passing with Ports
- Relational Programming

Advanced Oz

What is Oz?

Oz is a programming language

- conceived in 1991 by Gert Smolka at Saarland University, and
- subsequently developed in collaboration with Seif Haridi and Peter van Roy at SICS.

Oz is an experimental language and draws from experience in programming languages such as

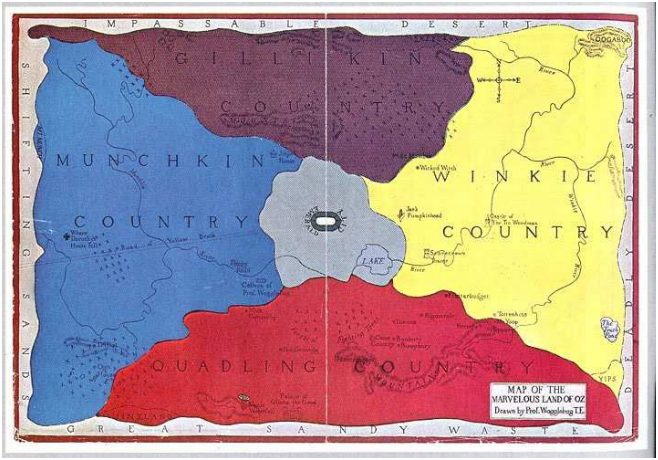
- Prolog,
- Erlang,
- LISP/Scheme, etc.

Why Oz?

Oz is a multiparadigm PL and includes features such as

- imperative (stateful) and functional (stateless) programming;
- data-driven (eager) and demand-driven (lazy) execution;
- relational (logic) programming and constraint-propagation;
- concurrent and distributed programming,
- object-oriented programming.

This makes Oz an interesting language for teaching and research.


IMPASSABLE DESERT
GILLIKIN COUNTRY
MUNCHKIN COUNTRY
WINKIE COUNTRY
QUADLING COUNTRY
SHIFTING SANDS
DEADLY DESERT
GREAT SANDY WASTE
LAKE
OOGABOO
YIPS
MAP OF THE MARVELOUS LAND OF OZ
Drawn by Prof. Wogglebug T.E.

Installing Oz

Oz is an interpreted and/or compiled language, implemented in the Mozart platform. To install Mozart,

- go to http://www.mozart-oz.org/download,
- choose the installation package relevant for your platform,
- follow the instructions.

Mozart is pretty well documented, so you should have no problems.

- If all goes well, you should be able to start Mozart and see a message like:

```
Mozart Compiler 1.4.0 (20090502013126) playing Oz 3
```

Playing Oz: say ‘Hello!’

- open the Oz Programming Interface (OPI);[1]
- type `{Browse 'Hello!'}` in the program buffer;
- type C-. C-b to execute the program.[2]

If all goes well, a Browser window should pop up with `'Hello!'` written in it.

- You’ve executed your first Oz program in the interactive mode.
- The syntax `{...}` denotes application of a function.
- `Browse` is a variable with a function value.
- `'Hello!'` is an atom (roughly, a constant).

[1] E.g., type `oz &` on the command line.
[2] ‘C-. . .’ stands for ‘control and . . .’. Alternatively, type M-x (Alt-x) followed by `oz-feed-buffer`.

Oz code can also be compiled into a command-line executables.[3]

- Save the following code into `hello.oz`.

Example

`hello.oz`

```
functor
import
   Application
   System
define
   {System.showInfo "Hello!"}
   {Application.exit 0} end
```

[3] The compiled code is not native binary, but a shell script-wrapper with embedded Oz virtual machine bytecode.

Playing Oz *contd*

Playing Oz: say 'Hello!' again

- compile `hello.oz` into an executable, and execute it:

```
$ ozc -x hello.oz
$ ./hello
Hello!
```

The 'binary' is implicitly executed on the Oz virtual machine, the Oz engine.[4]

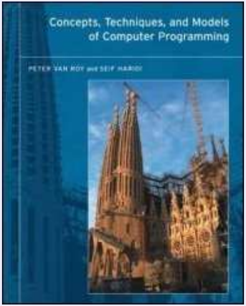
- Instead of opening a separate browser window, the output is sent directly to the standard output stream.

[4] The Oz VM can also be invoked explicitly as `ozengine hello`.

Playing Oz *contd*

More on Oz

Concepts, Techniques and Models of Computer Programming, 1st ed.
P. van Roy, S. Haridi, MIT Press 2004



... and online docs.

Lecture Outline

Introduction to Oz
- Oz & Mozart
- Playing Oz

Programming in Oz: Basics
- `mdc`, the Little Brother

Programming Oz: More Features
- Last-Call Optimization and Tail-Recursion
- Dataflow Variables
- Concurrency, Streams, Synchronization
- Lazy Evaluation
- Message Passing with Ports
- Relational Programming

Advanced Oz

Programming (not specifically in Oz)

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots.

So far, the Universe is winning.

Rick Cook

If debugging is the art of removing bugs, then programming must be the art of inserting them.

(anonymous)

Consider the task of implementing a simple, stack-based command line calculator.

- mdc,[5] the calculator, will have a simple postfix syntax, and
- very limited functionality: basic arithmetic on integers.

Example

```
$ ./mdc -e '10 1 + p' # push 10 and 1, add, print
11
$ ./mdc <<END # likewise, using a two-line here-doc
> 10 1
> + p
> END
11
```

[5] 'mdc' stands for 'mini desktop calculator', a trivial clone of dc, a standard tool on many platforms.

`mdc`, the Little Brother *contd*

`mdc` programs push numbers on the stack, use arithmetic to combine them, and use the command `p` to print the top of the stack.

Running `mdc`

When called, `mdc` must perform a number of steps:

- read the input (from stdin, a file, or an option string);
- lexemize and tokenize the input;
- interpret the input, performing
- internal operations and IO (output), as necessary.

We shall see how (some of) these steps can be implemented in Oz.

Lexemization

The input is a string, a sequence of characters.

- We need to split it up into lexemes.
- Assume white space is a separator, everything else is lexemes.[6]

In our Oz implementation, we will use strings and lists of strings.

Example

input: `"10 1 + p"`

output: `["10" "1" "+" "p"]`

[6]For your own languages, don't write syntax specifications like this!

Let's implement a function that given a string returns a list of lexemes.

Example

mdc-lexemize.oz

```
fun {Lexemize Input}
   [String] = {Module.link ['x-oz://system/String.ozf']} in
   {String.split
    {String.strip Input unit} unit} end
```

`Lexemize`, a function of a single input (a string),

- binds a standard module to the local variable `String`;
- uses from the module to trim and split the string into a list.[7]

[7] In both cases, the atom `unit` means all white space is rubbish.

Tokenization

The input is a list of strings.

- We need to classify them as tokens.[8]

We will represent tokens as records – tagged tuples.

Example

input: `["10" "1" "+" "p"]`

output: `[int("10") int("1") op("+") cmd("p")]`

Each lexeme is wrapped into a record.

- The record's name is a constant representing the token's class.

[8] The lexeme-token terminology varies.

Example

mdc-tokenize.oz

```
fun {Tokenize Lexemes}
   case Lexemes of nil then nil
   [] Lexeme|Lexemes then Token in
      if Lexeme == "p" then Token = cmd(Lexeme)
      elseif {Member Lexeme ["+" "-" "*" "/"]}
      then Token = op(Lexeme)
      else Token = int(Lexeme) end
      Token|{Tokenize Lexemes} end end
```

`Tokenize`, a function of a list of lexemes,

- uses pattern-matching to decompose the input;
- classifies and correspondingly wraps the first lexeme;
- constructs a list of tokens, calling itself recursively with the rest of lexemes.[9]

[9] The base case being the empty list, of course.

mdc, the Little Brother *contd*

We can simplify the code a little bit by using higher-order programming.

Example

mdc-tokenize.oz

```
fun {Tokenize Lexemes}
   {Map Lexemes
    fun {$ Lexeme}
       if Lexeme == "p" then cmd(Lexeme)
       elseif {Member Lexeme ["+" "-" "*" "/"]} then op(Lexeme)
       else int(Lexeme) end end} end
```

- Tokenize maps an anonymous function onto every element in Lexemes.

`mdc`, the Little Brother *contd*

Parsing, Compilation, Interpretation

The `mdc` language is trivially simple.

- There is virtually no need for parsing.
- We can skip compilation and interpret directly the sequence of tokens, one token at a time.

Example

input: `["10" "1" "+" "p"]`

interpretation: push(10)
push(1)
push(add(pop(), pop())
print()


GO, MY SON...
MAY ALL YOUR BUGS
BE COMPILE-TIME, AND
ALL YOUR GRAMMARS BE
CONTEXT-FREE!

Example

mdc-interpret.oz

```
proc {Interpret Tokens}
   proc {Interpret Stack Tokens}
      case Tokens of nil then skip
      [] int(Lexeme)|Tokens then
         {Interpret {String.toInt Lexeme}|Stack Tokens}
      [] op(Lexeme)|Tokens then Int1|Int2|Rest = Stack
         Operator = try Number.{String.toAtom Lexeme}
                    catch _ then Int.'div' end in
         {Interpret {Operator Int2 Int1}|Rest Tokens}
      [] cmd("p")|Tokens then Top|_ = Stack in
         {Browse Top} {Interpret Stack Tokens} end end in
   {Interpret nil Tokens} end
```

`Interpret` is a procedure that

- iteratively processes a list of tokens,
- modifying the stack as necessary;
- `try ... catch` is used to retrieve the appropriate arithmetic function.

Wrap-up

We've used some of the basic functionalities in Oz:

- list processing,
- pattern matching,
- higher-order programming,
- exceptions.

Let's compile and execute `mdc`.[10]

```
$ ozc -x mdc.oz
$ ./mdc -e '1 2 + p' # or ./mdc <<< '1 2 + p'
$ ./mdc test.mdc # or ./mdc -f test.mdc, or ./mdc < test.mdc
$ echo '1 2 + p' | ./mdc
```

[10] The file `mdc.oz` defines an executable functor that imports our partial implementation files seen on previous slides.

Lecture Outline

Introduction to Oz
- Oz & Mozart
- Playing Oz

Programming in Oz: Basics
- `mdc`, the Little Brother

Programming Oz: More Features
- Last-Call Optimization and Tail-Recursion
- Dataflow Variables
- Concurrency, Streams, Synchronization
- Lazy Evaluation
- Message Passing with Ports
- Relational Programming

Advanced Oz

Last-Call Optimization and Tail-Recursion

Last Call-Optimization

Oz supports tail-recursive procedures.

Factorial is defined recursively in terms of itself:

$$n! = \begin{cases} 1 & \text{if } n < 2 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

Example

factorial.oz

```
fun {Factorial N}
   fun {Factorial N F}
      if N < 2 then F
      else {Factorial N-1 F*N} end end in
   {Factorial N 1} end
```

`Factorial` runs with $O(1)$ frame stack.[11]

[11] And is thus immune to the ‘maximum recursion depth exceeded’ error.

Dataflow Variables

Dataflow Variables

Oz variables are not variable. A variable can

- be unbound (have no value),
- become bound to a particular value for the rest of its lifetime.

In Oz,

- variable identifiers are lexically scoped, and
- variables are not assigned to, but rather unified.

Example

```
local X in      % X is introduced and is unbound
   X = 1        % X is unified with 1
   X = 2        % X is already 1, unification failure
   ... end
```

Dataflow Variables *contd*

Example

```
local X Y in    % X and Y are introduced and are unbound
   X = Y        % X is unified with Y, both are still unbound
   X = 1        % both X and Y have the value 1
   ... end
```

Some operations on unbound variables cause blocking.

Example

```
local X Y in
   X = Y + 1    % can’t bind X until Y is bound; halt
   ... end
```

```
local X in
   if X == 0    % can’t test X it is bound; halt
   ... end
```

Dataflow Variables *contd*

Unification

When variables are unified, their values

- are compared for equality, or
- are established, if necessary.

Unification works also on data structures of arbitrary complexity.[12]

Example

```
local
   X Y
   fun {Tree Left Right}
      t(l:Left r:Right) end in
   t(r:X l:t(l:0 r:1) = {Tree Y t(l:2 r:3)}
   % X is t(l:2 r:3), Y is t(l:0 r:1)
end
```

[12] We've seen it in action while decomposing the stack in `mdc`.

Concurrency and Synchronization

Unbound variables may be used to synchronize threads of computation.

Let's have

- a producer produce, and
- a consumer consume

an infinite stream.[13]

If the consumer is slower than the producer, the latter will be wasting resources (time, space, power).

- We can solve this roblem by having the producer wait for demand from the consumer.

[13] Infinite streams in this context are simply endlessly growing lists.

Example

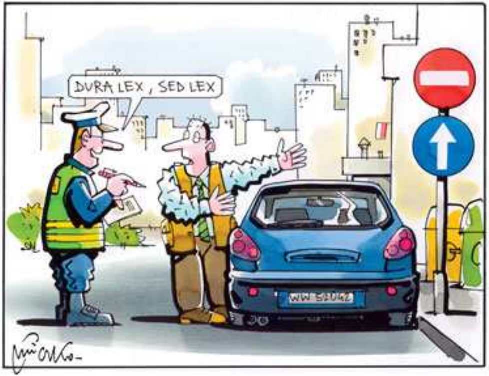
enumerate.oz

```
fun {Enumerate N}
   proc {Iterate N Request|Rest}
      Request = N
      {Iterate N+1 Rest} end in
   thread {Iterate N $} end end
```

map.oz

```
fun {Map Stream Function}
   fun {Iterate Stream}
      Head|Tail = Stream in
      {Function Head}|{Iterate Tail} end in
   thread {Iterate Stream} end end
```

Both `Enumerate` and `Map` start separate threads of computation, but when coupled, their progress depends on each other.


DURA LEX, SED LEX

Example

```
local Integers Squares in
   Integers = {Enumerate 0}
   Squares = {Map Integers fun {$ N} N*N end} end
```

1. `{Enumerate 0}` starts a new thread, but no integers appear until the call to `Map` is executed.
2. When `Map` starts a new thread, it places an unbound variable (a `Head`) on the `Integers` stream, and halts until it becomes bound.
3. `Enumerate` reactivates, binds the `Request` variable, and proceeds with the rest of `Integers`—which is unbound and halts `Enumerate` again.
4. With `Head` bound, `Map` reactivates, places `{Function Head}` on the output stream `Squares`, and places a new request on `Integers`.
5. `GOTO 3`

Infinite streams can be useful for generating. . . further infinite streams.

Being One's Own Tail

Everyone knows what a Fibonacci number is.[14]
What is the infinite stream of Fibonacci numbers?

- It is the first two numbers followed by the stream of Fibonacci numbers added to itself left-shifted by one.

This trivially translates to code. . .

Example

fibs.oz

```
Fibs = 1|1|{Add Fibs {Drop Fibs 1}}
```

[14] If you don't, it is the sum of the previous two Fibonacci numbers, save for the first two 1's.

We only need `Add` and `Drop` for this to work.

Example

add.oz

```
fun {Add Stream1 Stream2}
   case Stream1#Stream2 of (Head1|Tail1)#(Head2|Tail2)
   then thread Head1+Head2 end|{Add Tail1 Tail2} end end
```

drop.oz

```
fun {Drop _|Tail N}
   if N == 1 then Tail
   else {Drop Tail N-1} end end
```

- `Add` decomposes the streams into their heads and tails, adds the former (in a separate thread!)[15] and places on top of the recursively computed sum of the latter.
- `Drop` recursively skips elements from the stream, as needed.

[15] Yes, it can add streams of yet unbound variables.

Actually, we do not need all this hassle to implement infite streams.

- Guess what, it suffices to be lazy.

Example

fibs-lazy.oz

```
Fibs = local fun lazy {Fibs PrePrevious Previous}
                Current = PrePrevious + Previous in
                Current|{Fibs Previous Current} end in
          1|1|{Fibs 1 1} end
```

- `Fibs` is a local function that, given two numbers, lazily places their sum on top of the result of a call to itself.
- `Fibs` produces, in principle, and endless stream—but only as much of it as needed.[16]

[16]The core issue is to define ‘needed’.


HARD WORK PAYS
OFF IN THE FUTURE
LAZINESS PAYS OFF
NOW
© Line Maro

Message Passing with Ports

The style of communication between consumer and producer we've seen previously was inconvenient:

- explicitly adding unbound variables to the stream is both cumbersome and error-prone;
- only one producer can be extending a particular stream.[17]

Message Passing

Ports provide a convenient abstraction for asynchronous between-thread communication via message passing.

- A port is bound to a stream.
- Any thread can send a message to the port.
- Messages sent to a port are placed on the stream in a partially specified order.

[17] There are workarounds, but they're even more cumbersome.

Example

receive.oz

```
proc {Receive Messages React}
   thread {ForAll Messages React} end end
```

`Receive` is a procedure that

- given a (potentially infinite) stream of messages,
- applies a particular procedure to the messages, one by one, in a dedicated thread.

Example

```
{Receive Messages
 proc {$ Message}
    {Browse received(Message.content)} end}
```

Example

`spam.oz`

```
proc {Spam Port Message}
   proc {Repeat}
      {Delay {OS.rand} mod 1000}
      {Send Port Message}
      {Repeat} end in
   thread {Repeat} end end
```

`Spam` is a procedure that

- given a port and a message,
- repeatedly sends the message to the port, in random intervals, in a dedicated thread.

Example

```
{Spam Port msg(priority:urgent content:'end the lecture')}
```

Example

filter.oz

```
fun {Filter Messages Pass}
   fun {Filter Message|Messages}
      if {Pass Message} then Message|{Filter Messages}
      else {Filter Messages} end end in
   thread {Filter Messages} end end
```

`Filter` is a procedure that

- given a stream of messages and a Boolean test function,
- filters out those messages that do not pass the test.

Example

```
{Filter Messages
 fun {$ Message}
    case Message of msg(priority:Priority ...)
    then Priority /= urgent end end}
```

Message Passing with Ports *contd*

We can actually ship unbound variables, data, and procedure objects between different Oz processes.

Example

server.oz

```
fun {Server Handle Ticket}
   Requests Port = {NewPort Requests} in
   {Pickle.save
    {Connection.offerUnlimited Port}
    Ticket}
   server(start:proc {$}
                   thread {ForAll Requests Handle} end end) end
```

Server

- creates a stream and a port,
- opens up access to the port through a connection,
- returns a (wrapped) procedure that will spawn a thread for handling the arriving messages.

Example

client.oz

```
fun {Client Generate Ticket}
   Port = {Connection.take
           {Pickle.load Ticket}}
   proc {Loop}
      {Send Port {Generate}}
      {Loop} end in
   client(start:proc {$}
                    thread {Loop} end end) end
```

`Client`

- accesses a port through a connection,
- creates a procedure that will repetitively send messages to the port,
- returns a (wrapped) procedure that will spawn a thread for actually sending the messages.

Message Passing with Ports *contd*

Example

```
{{Server
  proc {$ msg(Request Process Response)}
     {Browse request(Request)}
     Response = {Process Request} end
  Ticket}.start}
{{Client
  proc {$}
     Response in
     {Browse response(Response)}
     msg({OS.rand} mod 100 fun {$ N} N mod 10 Response) end
  Ticket}.start}
```

- The client sends messages containing a number, a function, and an unbound variable.
- The server applies the function to the number and binds the variable to the result.

Consider the problem of appending one list to another.

Example

append.oz

```
proc {Append Front Rear Whole}
   case Front of nil then Whole = Rear
   [] Head|Tail then WholeTail in
      Whole = Head|WholeTail
      {Append Tail Rear WholeTail} end end
```

Does it work relationally?

Example

```
{Browse {Append [1 2] [3 4] $}}     % [1 2 3 4]
{Browse {Append [1 2] $ [1 2 3 4]}} % [3 4]
{Browse {Append $ [3 4] [1 2 3 4]}} % ?
```

Relational Programming

Consider the problem of appending one list to another.

Example

append.oz

```
proc {Append Front Rear Whole}
   case Front of nil then Whole = Rear
   [] Head|Tail then WholeTail in
      Whole = Head|WholeTail
      {Append Tail Rear WholeTail} end end
```

Does it work relationally?

Example

```
{Browse {Append [1 2] [3 4] $}}     % [1 2 3 4]
{Browse {Append [1 2] $ [1 2 3 4]}} % [3 4]
{Browse {Append $ [3 4] [1 2 3 4]}} % ?
```

- In the last case, pattern matching blocks over the unbound variable `Front`.

However, instead of the blocking pattern matching, we can

- make perform a series of unifications in different search spaces, and
- choose the ones that succeed.

Example

append-or.oz

```
proc {Append Front Rear Whole}
   or Front = nil
      Rear = Whole
   [] Head FrontTail WholeTail in
      Front = Head|FrontTail
      Whole = Head|WholeTail
      {Append FrontTail Rear WholeTail} end end
```

```
{Browse {Append $ [3 4] [1 2 3 4]}} % [1 2]
```

We can have even more fun with `Append` if we replace `or` with `choice`.

Example

```
{Browse
 {SolveAll
  fun {$}
     Front Rear in
     {Append Front Rear [1 2 3 4]}
     sol(Front Rear) end}
% [sol(nil [1 2 3 4])
%  sol([1] [2 3 4])
%  sol([1 2] [3 4])
%  sol([1 2 3] [4])]
%  sol([1 2 3 4] nil)]
```

- Much like Prolog, no?

Lecture Outline

- Introduction to Oz
  - Oz & Mozart
  - Playing Oz
- Programming in Oz: Basics
  - `mdc`, the Little Brother
- Programming Oz: More Features
  - Last-Call Optimization and Tail-Recursion
  - Dataflow Variables
  - Concurrency, Streams, Synchronization
  - Lazy Evaluation
  - Message Passing with Ports
  - Relational Programming
- **Advanced Oz**

Advanced Oz

We have barely touched the surface, there's a lot more!

- finite domain constraint programming;
- distributed programming;
- object-oriented programming;
- shared-state sequential and concurrent programming;
- system programming; etc.

Check out the docs, have fun.

Thank you!