Seminar Report

The Oz Programming Language

Benjamin Vettori
benjamin.vettori@student.uibk.ac.at

31 July 2015

Supervisor: Andreas Kochesser

Abstract

The Oz programming language is a multi-paradigm programming language which is quite unknown, although it has some interesting aspects. This seminar report gives an overview of the Oz programming language supported by code-samples and some features of Oz will be compared to other similar programming languages.

Contents

1 History **1**

2 Dedicated purpose of Oz **2**

3 Introduction to Oz **2**
3.1 Basics . . . 2
3.1.1 Types in Oz . . . 2
3.1.2 Procedures, Functions and Scoping . . . 5
3.2 Parallel Programming . . . 7
3.2.1 Futures . . . 7
3.3 Distributed Programming . . . 8
3.4 Constraint Programming . . . 9
3.4.1 The Constraint Store . . . 9
3.4.2 Computation Spaces . . . 9
3.4.3 Send More Money Example . . . 10

4 Highlights of the Oz Programming Language **11**
4.1 Logic Variables and Concurrent Control . . . 11
4.2 Arbitrary long Integers . . . 12
4.3 The Oz Computation Model . . . 12

5 Differences to other similar Programming Languages **12**

6 Conclusion **13**

Bibliography **14**

Introduction

The Oz programming language is a so called multi-paradigm programming language, which was an attempt to create a better programming language for parallel constraint programming. The first part of this report will start with a rough overview about the history of the development of Oz and describe the dedicated purpose of Oz. Then in Chapter 3 an introduction to the basics of Oz is given and the paradigms parallel, distributed and constraint programming will be explained in more detail with some code samples. Chapter 4 illumines features of Oz which are unique in Oz or rather rare in other languages. Some features of Oz will be compared to other (similar) programming languages and there is a conclusion given to wrap up the seminar report.

1 History

This section is concerned with the history of the Oz programming language. The years mentioned are mainly based on the publication dates of Oz related papers to get an approximate timeline for the progress of the Oz programming language.

The Oz programming language was originated by Gerd Smolka and his team at the Center for Artificial Intelligence in the University of Saarbruck. Oz has been published in 3 versions (Oz1, Oz2 and Oz3). The major difference from Oz1 to Oz2 are in the threading model they used, this will be explained in chapter 3.2. The major additional features of Oz3 compared to Oz2 are functors and futures. Functors are comparable to Java packages. They can be imported as module and encapsulate code. Futures are used for lazy evaluation of data, so the value of a future is evaluated when it gets accessed. The DFKI Oz system has been redeemed by the Mozart system, which was available in a stable version (1.4) and an alpha Version (2.0) to the time this report was written. (note: all examples in this report are written in Oz3).

1990 They started with "[...] concurrent constraint programming, which brings together ideas from constraint and concurrent logic programming" [3].

1993 The first paper mentioning the language Oz was published: "Oz - A Programming language for Multi-Agent Systems" [3].

1995 Introduction of object-oriented concurrent constraint programming [2].

1999 The Mozart Consortium, consisting of Saarland University, Swedish Institute of Computer Science (SICS) and the Catholic University of Louvain continues with Oz development from now on.

2000 The Oz Browser has been introduced, a graphical tool to display Oz data structures in a dynamic and efficient way [1]

2002 The Integrated Oz Search Factory (IOzSeF) has been introduced, a "[...] library which provides a generic interface to search in constraint programming systems" [7].

2 Dedicated purpose of Oz

When project Hydra was started in the year 1991 at the DFKI, they had the goal to create "[...] a high-level concurrent programming language bringing together the merits of logic and object-oriented programming in a unified language" [3]. They wanted to create a higher-order language to achieve the integration of constraint and object oriented programming and they needed another communication model for concurrent logic programming, since they found that communication based on streams was not compatible with their goals, because "[...] it induces a tedious and low-level programming style [...] and it poses serious implementation problems due to the need for fair stream merging [...]" [5].

Oz combines the advantages of concurrent, logic-/constraint, functional and object oriented programming. It "[...] is designed as a successor to languages such as Lisp, Prolog and Smalltalk, which fail to support applications that require concurrency, reactivity and real-time control" [6]. They actually reached their goals since there is an implementation (Mozart) for the Oz programming language version 3, which supports concurrent constraint programming, class based object-orientation, communication through ports and more additional features. There are just a few languages which are capable to support as much programming paradigms as Oz does.

3 Introduction to Oz

This chapter shall give the reader a shallow introduction to the Oz programming language.

3.1 Basics

In this section the Basics about the Oz programming language will be explained. It gives a basic overview over the types, functions and procedures in Oz.

3.1.1 Types in Oz

Figure 1 shows the type system of the Oz programming language. Some types are known from other languages, so they are explained in a shorthand style in table 1, while uncommon types like chunk, cell and space are explained in more detail.

Type Conversion

There is no implicit type conversion in Oz, but the system modules provide type conversion procedures like `{Int.toFloat I ?F}` and `{Float.toInt F ?I}`. If two different types are used for a numeric operation (i.e. Int + Float), an exception is thrown.

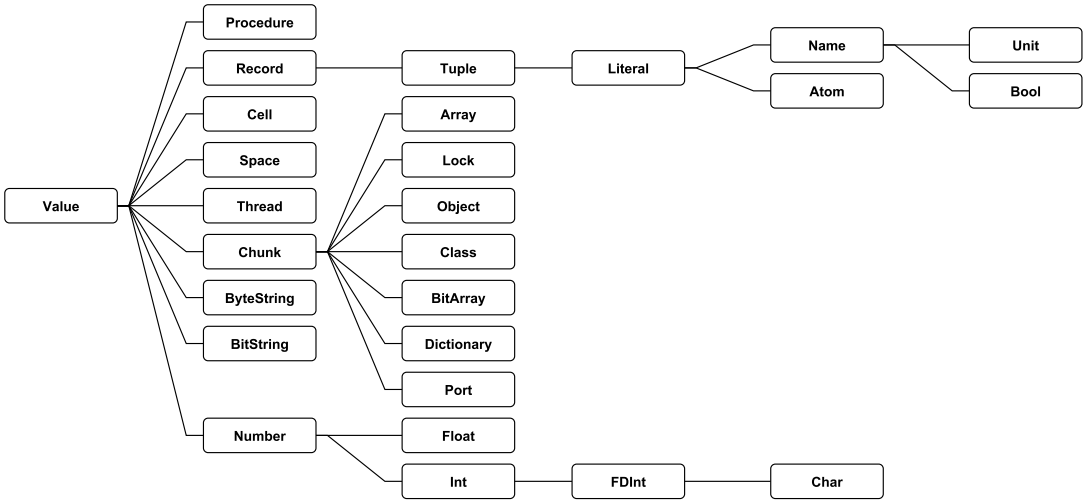

Figure 1: Type system of the Oz programming language

FDInt

FDInt stands for Finite Domain Integer, they are used for constraint programming in Oz. FDInt's may have values in the range from 0 to 2^{27}-2 (i.e., 134217726)

Record, Tuple and Literal

Records

A record has a label and a fixed number of components. An example for a record is `tree(key: I value: Y left: LT right: RT)`, where 'tree' is the label of the record and 'key', 'value', 'left' and 'right' are features of the record with values stored in variables I, Y, LT and RT.

Tuples

If the component names of a record are omitted, this is called a tuple. In this case the component names of the record are just the natural numbers starting with 1. So a tuple variant of the above record is `tree(1:I 2:Y 3:LT 4:RT)`. Oz provides a syntactic sugar notation for tuples where the programmer may omit the features: `tree(I Y LT RT)`

Literals

Literals are atomic types. They are divided into atoms and names. Atoms are a number of alphanumerical letters where the first letter is lowercase or arbitrary printable characters enclosed in quotes (i.e. **helloWorld** and **'HELLO@WORLD'** are atoms). Names can

Type	Operations	Examples
Number	+, –, *, ~ (unary minus), *pow* ($C = A^B$) and *abs* ($B = \|A\|$)	$A = B + C$, $\{Number.pow\ A\ B\ C\}$
Int	*is*, *isNat* ($A \geq 0$), *isOdd*, *div* ($C = A/B$), *mod*, *toFloat*, ...	$\{Int.div\ A\ B\ C\}$, $\{Int.is\ A\ B\}$ (B is true if A is an integer)
Float	/, *is*, *exp*, *log*, *sqrt*, *ceil*, *floor*, ...	$A = B\ /\ C$, $\{Float.exp\ A\ B\}$ ($A = e^B$)
Char	*is*, *isLower*, *toLower*, *isUpper*, *toUpper*, *isAlNum*, ...	$\{Char.toUpper\ A\ B\}$ ($A = `a` \implies B = `A`$)

Table 1: Table caption

be generated with the `{NewName X}` procedure. The `X` which is passed to the `NewName` procedure is a variable and the value which gets assigned to X is a new worldwide unique name. The boolean values **true** and **false** are special names and **unit** is a special name that is mainly used as synchronization token of concurrent programs.

Chunk and Cell

Chunks

Chunks are used for abstract representations of data structures. There is no arity operation on chunks, that means some of their components can be hidden (i.e. private members of classes).

Cells

Cells are a special type of chunks. They can be seen as chunks with a mutable component. While variables in Oz are not mutable, cells can be used to write imperative programs. The '@' operator can be used to get the value of a cell and the ':=' operator modifies a cell's content. Listing 1 shows an Oz program using cells to sum up the Numbers from 1 to 10.

```
local J in
  J = {NewCell 0}
  {For 1 10 1 proc {$ I}
    J:=@J+I
  end}
  {Browse @J}
end
```

Listing 1: Imperative program using cells to sum up the numbers from 1 to 10

Space

A (computation) space is a container used in constraint programming. This type is explained in more detail in section 3.4.2

Strings

In Oz strings are just arrays of characters which are integers from 0 to 255, so the list [79 90 32 51 46 48] equals [&O &Z &␣ &3 &. &0] which equals "OZ 3.0". Oz provides so called virtual strings, which are tuples with label '#'. `VirtualString` is a system module which provides methods to concatenated the values of a virtual string into one single string. Listing 2 provides an example for the use of a virtual string.

```
declare X = "Hello␣"#"world!"
declare Y = atom#1#"␣test␣"#1.0
declare Z = X#"␣"#Y
{Show {VirtualString.toByteString Z}}
%shows <ByteString "Hello world! atom1 test 1.0">
```

Listing 2: Example program for the use of virtual strings

3.1.2 Procedures, Functions and Scoping

Procedures

Procedures in Oz are very similar to procedures in Ocaml. A procedure can be written as given in listing 3. A procedure has no return value, so if a return value is desired, it has to be assigned to a passed argument. Listing 3 shows two samples how a return value of a procedure can be used (assuming the third parameter is used as return value in the procedure).

```
declare proc {ProcedureName Param1 Param2 ...}
/* do something here */
end
% use the thirt parameter as return value and assign it to X
declare X = {ProcedureName P1 P2 $}
% declare a variable Y and use it as return value for the procedure
declare Y;
{ProcedureName P1 P2 Y}
```

Listing 3: Procedures in Oz

Functions

Functions are syntactic sugar for procedures, where an to the programmer ”invisible” argument is used as return value and the last statement in a function is assigned to this invisible variable. Listing 4 shows the syntax for a function and an example assignment of the result.

```
declare fun {FunctionName Param1 Param2 ...}
        /* do something here */
        /* the last statement is returned */
end

% assign the return value of the function to X
declare X = {FunctionName P1 P2 ...}
```

Listing 4: Functions in Oz

Scoping

It is possible to introduce local variables using the `local` keyword. A local is equivalent to Ocaml's `let ... in ... ;;` construct. Listing 5 shows an example use for a local area, where the variable `X` declared between the keywords `local` and `in` can only be used within the local. There is a syntactic sugar variant for locals in procedures and functions, where the `local` and `end` keywords can be omitted. Listing 6 shows an example for such a local within a procedure.

```
local X in
  /* do something with X herel */
end
```

Listing 5: Local in Oz

```
declare proc {Proc1 X}
  X
in
  /* use X here, X is local */
end
```

Listing 6: Local within a procedure

3.2 Parallel Programming

Parallel programming in Oz1 is quite different to its successors. While in Oz1 every statement could be executed in parallel, in Oz2 and later one has to spawn a thread explicitly. While the automatic thread spawning was theoretically appealing, it was very hard for the programmer to write such programs. Thus they decided to make thread spawning an explicit statement. An example for a parallel execution in Oz1 is given in listing 7 and listing 8 shows the equivalent program for Oz2 and later.

```
declare X Y in
  % X = 2+2 and Y = 4+4 is executed in parallel
  {Add 2 2 X} {Add 4 4 Y}
end
```

Listing 7: Example for a parallel execution in Oz1

```
declare X Y in
  % X = 2+2 and Y = 4+4 is executed in parallel
  thread {Add 2 2 X} end
  thread {Add 4 4 Y} end
end
```

Listing 8: Example for a parallel execution in Oz2 and later

3.2.1 Futures

In Oz3 they introduced Futures as a concept for lazy evaluation of values. A future can be seen as object, whose value gets only evaluated when it is accessed. The `NumGen` function in listing 9 works well for finite numbers. To get infinite many numbers the `NumGen` function would have to create an infinitely large array, which is not possible. To solve this problem, a lazy number generator could be written, using the advantage of futures. Listing 10 shows an example for a number generator which produces an infinite stream of numbers starting with the given parameter, but the values are always just created when they get accessed. This is done by the call of the `ByNeed` primitive operation, which creates a future for the passed function. The $-sign is used to take the second parameter (the future) as return value, so that the result of `NumGen` is a list with one value as head and a future as tail, which gets evaluated when it gets accessed.

```
declare fun {NumGen X Limit}
  case X < Limit then X|{NumGen X+1 Limit}
    else nil
  end
end
```

Listing 9: A finite number generator

```
declare fun {NumGen X}
  X|{ByNeed {NumGen X+1} $}
end
```

Listing 10: A lazy number generator

3.3 Distributed Programming

Oz provides system libraries for distributed programming. The concept used for transparency of the distribution of objects, functions, variables, etc. is explained in detail in [8]. The `Pickle` module can be used to export and import objects such as procedures, classes, etc. into and from files. This allows to make them publicly available to others, while the access restrictions are handled by other programs (file-system, web-server, ...). The `Connection` module allows to handle the type of the connection used to offer an object. The procedures `Offer` and `Take` in listing 11 are a small abstractions which allow to save a file with an object, offered unlimited times. This means it can be taken as long as the process offering this object is alive. The `Take` procedure on the other hand allows to load an object via URL, which could just be the URL to a file at a web-server like *'http://localhost/myService/sumList'*. Listings 12 and 13 show an example for a producer and a consumer using a distribution. The consumer offers its `Xs` variable as *'http://localhost/myService/sumList'* to a local webserver. The producer loads the remote object `Xs` with the `Take` procedure, produces a list with numbers from 0 to 150000 and writes them to `Xs`. Since `Xs` gets passed to the Sum function at the consumer, the Values of Xs get summed up, when the producer assigns the list to `XS`. The `Sum` function sums up the list elements until `Xs` equals `nil`, and writes the result in the `Browser`. Here the behavior of Oz variables is exploited, since the consumer thread blocks until `Xs` is assigned to a value, which is in this case the list of numbers produced by the producer. This is just a simple example for distributed programming, but since it is also possible to i.e. pass around procedures or classes, one could easily write a server and client program, where each client can load procedures to be calculated from the server, runs them and writes the result back to the server.

```
proc {Offer X FN}
  {Pickle.save {Connection.offerUnlimited X} FN}
end

proc {Take FNURL}
  {Connection.take {Pickle.load FNURL}}
end
```

Listing 11: Offer and Take procedures for distributed programming. [8]

```
declare Xs Sum in
  {Offer Xs 'http://localhost/myService/sumList'}
  fun {Sum Xs A}
    case Xs of X|Xr then {Sum Xr A+X}
    [] nil then A
    end
  end
  {Browse {Sum Xs 0}}
end
```

Listing 12: Sum procedure offered to others (consumer). [8]

```
declare Xs Generate in
  Xs={Take 'http://localhost/myService/sumList'}
  fun {Generate N Limit}
    if N<Limit then N|{Generate N+1 Limit} else nil end
  end
  Xs={Generate 0 150000}
end
```

Listing 13: Producer for a list of numbers: N, N+1, ... , Limit. [8]

3.4 Constraint Programming

3.4.1 The Constraint Store

The constraint store is a component which is shared across all Oz threads and can be used to store variable bindings in form of equalities. "Conceptually the store is modeled as a conjunctive logical formula: $\exists Y_1 \ldots Y_m : X_1 = U_1 \wedge \ldots \wedge X_n = U_n$, where the X_i are the variables and U_i are Oz values or variables, and Y_i are the union of all variables occuring in X_i and U_i" [4]

3.4.2 Computation Spaces

Computation spaces consist of a computation store and a set of executing threads. A computation store consists of a constraint store, a procedure store and a cell store. There

are some general rules about computation spaces:

1. If a thread tries to add an inconsistent constraint to the store, an exception gets raised in the thread, so the constraint store of a computation space remains consistent.

2. Computation spaces may contain (child) computation spaces.

3. "A thread belongs always to one computation space. Also, variables belong to only one computation space." [4]

4. Threads in a space may access and bind variables of the space and of all ancestor spaces. A thread of an ancestor thread cannot access or bind variables of a child space.

3.4.3 Send More Money Example

Listing 14 shows an example program using constraint programming to solve the very well known logic puzzle SEND+MORE=MONEY. This shall give the reader an impression about constraint programming in Oz. The program can be explained as follows:

- In line 1 a tuple for the solutions of all single letters is created.

- The `:::` operator in line 3 is used to assign all letters (in the `Values` tuple) a range itself of integers (0..9), where the range is represented as a tuple with the lower and upper bound (0#9).

- The `FD.distinct` method, called in line 4, is a default library procedure which creates constraints to ensure that all values of the `Values` tuple are distinct.

- The `=:` and `\=:` operators are used to add equal- and not-equal-constraints to the store. In line 5 and 6 constraints get added, which ensure that the letters S and M are not equal to zero.

- In lines 7 and 8 a constraint gets added, which ensures the statement SEND + MORE = MONEY.

- In line 9 the solutions for the single letters in the `Values` tuple get resolved by the `FD.distribute` method with a first fail algorithm.

- In line 12 the `Search.base.all` method is called with the `Money` procedure. There are a couple of search procedures in Oz, the `all` procedure searches all valid results. In this case there exists just one result which will be displayed in the browser: $sol(d{:}7\ e{:}5\ m{:}1\ n{:}6\ o{:}0\ r{:}8\ s{:}9\ y{:}2) \implies 9567 + 1085 = 10652$.

```
1  proc {Money Values} S E N D M O R Y in
2    Values = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)
3    Values ::: 0#9
4    {FD.distinct Values}
5    S \=: 0
6    M \=: 0
7    (1000*S + 100*E + 10*N + D) +
8    (1000*M + 100*O + 10*R + E) =: 10000*M + 1000*O + 100*N + 10*E + Y
9    {FD.distribute ff Values}
10 end
11
12 {Browser.browse {Search.base.all Money}}
```

Listing 14: The send more money example

4 Highlights of the Oz Programming Language

This chapter points out some features of Oz, that are not very common in other languages.

4.1 Logic Variables and Concurrent Control

Variables in Oz behave like logic variables and are comparable to final variables in Java. They can be used in calculations, even if they are not assigned to a value, since a thread that reaches the variable suspends until a value is assigned to the variable. This is useful, since this means it is possible to do calculations as far as possible in parallel. Listing 15 shows an example, where the thread using variable `X` is waiting until a value is assigned to `X` by another thread. This is deterministic because of the behavior of variables in Oz.

```
declare X Y in
  /**
   * X = 2 and Y = X*X is executed in parallel, but the
   * second thread suspends until 2 is assigned to variable X
   */
  thread
    X = 2
  end
  thread
    Y = X*X
  end
end
```

Listing 15: Example for automatic suspending of threads by variables without value

4.2 Arbitrary long Integers

Oz provides arbitrary long integers. That means the size of an integer is just limited by the memory size. Integers with a size of up to 28 bits are stored in registers, if an integer is larger than 28 bit, it gets always loaded from main memory, so in terms of performance it is better to use multiple small integers instead of a large one.

4.3 The Oz Computation Model

Oz is based on a new computation model, which has been carefully designed and is based on the π-calculus. They discovered "new primitives for expressing computational abstractions such as functions, objects and search" [6]. More details about the Oz-calculus can be found in [5].

5 Differences to other similar Programming Languages

Oz has similarities to several other programming languages, since it implements several programming paradigms. The most object oriented programming languages use a class based object oriented style, as Oz does, so this is a very obvious similarity to other languages like Java, C++ and C#. There are only a few languages which support all programming paradigms that Oz supports. C++ for example supports all of them, if some libraries like Gecode for constraint programming and boolinq for declarative programming are used. But while there might always be another language which implements the same programming paradigm and has for some features similar syntax, Oz may behave different. Ocaml for example as shown in listings 16 and 17 has a similar pattern matching compared to Oz, but under the hood these two languages are completely different. Ocaml has custom types and mutable variables, while in Oz variables behave like logic variables and Oz is dynamically typed.

```
declare fun {Fac X}
  case X of 0 then 1
  [] X then X * {Fac X-1}
  end
end
```

Listing 16: Pattern matching in Oz

```
let rec Fac = function
  | 0 -> 1
  | n -> n * Fac (n-1);;
```

Listing 17: Pattern matching in Ocaml

Oz has also similarities to the programming language Prolog. According to the documentation "most Prolog programs have a straight forward translation to Oz programs" [4]. There are for example conditionals in Oz, which can be written as given in listing 18.

"The `cond` statement [...] corresponds roughly to Prolog's conditional `P -> Q ; R`. Oz is a bit more careful about the scope of variables, so local variables Xi have to be introduced explicitly. `cond X in P then Q else R end` always has the logical semantics $\exists X : P \wedge Q \vee (\nexists X : P) \wedge R$, given that we stick to the logical part of Oz. This is not always true in Prolog" [4].

```
cond X1 ... XN in S0 then S1 else S2 end
```

Listing 18: Conditional statement

6 Conclusion

This seminar report gave a shallow overview of the Oz programming language. Some features which are unique to Oz or uncommon in other languages, like the behavior of Oz variables or the constraint store, have been explained in more detail. All in all this report just scratched the surface of Oz, since there are numerous features which have not been explained. The developers of Oz have reached their goals, since they created the Mozart system, which implements Oz3 nearly completely. They were capable of creating a new language which brings together the merits of logic and object-oriented programming into one single language as a successor for languages such as Lisp or Prolog. Although the Oz programming language has an available implementation, there are just a few people who use this language in practice. Since functional and logic programming languages are not as widespread as procedural languages like C, it is unlikely that Oz, which supports multiple programming paradigms, will become a standard programming language within the next decades. Multiple programming paradigms means also multiple ways to program a specific behavior, what may result in bad maintainability. Thus it is unlikely that Oz can become a widespread programming language, but it may be interesting for teaching subjects like 'programming paradigms'. Peter Van Roy for example wrote a book about the 'Concepts, Techniques and Models of Computer Programming [8] which uses the Oz programming language as a single formalism for the presentation of all explained computation models and programs.

References

[1] T. Brunklaus. Der oz inspector - browsen: Interaktiver, einfacher, effizienter, feb 2000.

[2] M. Henz, G. Smolka, and J. Würtz. Object-oriented concurrent constraint programming in oz. 1993.

[3] M. Henz, G. Smolka, and J. Würtz. Oz—a programming language for multi-agent systems. In R. Bajcsy", editor, *13th International Joint Conference on Artificial Intelligence*, volume 1, pages "404–409", "Chambéry, France", "30 August–3 September" 1993. "Morgan Kaufmann Publishers".

[4] The Mozart Consortium. *Mozart Documentation*, 2008.

[5] G. Smolka. *A calculus for higher-order concurrent constraint programming with deep guards.* Deutsches Forschungszentrum für Künstliche Intelligenz, 1994.

[6] G. Smolka. An oz primer. In *DFKI Oz Documentation.* Citeseer, 1995.

[7] G. Tack. Iozsef - the integrated oz search factory. Technical report, 2002.

[8] P. Van-Roy and S. Haridi. *Concepts, techniques, and models of computer programming.* MIT press, 2004.