A Tutorial of Oz 2.0

Seif Haridi
SICS*

November 28, 1996

Abstract

We introduce in this tutorial the programming language Oz 2.0. Oz 2.0 is a multiparadigm programming language. It enables users to write significant applications in a fraction of time compared to other existing languages. The document is deliberately informal and thus complements other Oz 2.0 documentations.

1 Introduction

A very good starting point is to ask why Oz? Well, one rough short answer is that, compared to other existing languages, it is magic! It provides the programmer (or using the nicer term: application developer) with a wide range of programming/system abstractions to develop quickly and robustly advanced applications. And, yet it is a simple and coherent design. Oz tries to merge several directions of programming language designs into a single coherent one. Most of us know the benefits of the various programming paradigms whether object-oriented, functional or logic (constraint) programming. Yet when we start writing programs in any existing language, we quickly find ourselves confined by the concepts of the underlying paradigm. Oz tries to attack this problem by a coherent design of a language that combines the programming abstractions of various paradigms in a clean and simple way.

So, before answering the above question, let us see what Oz is? This is again a difficult question to answer in a few sentences. So here is the first shot. It is a high level programming language that is designed for modern advanced, concurrent, intelligent, networked, soft real-time, parallel, interactive and pro-active applications. As you see it is still hard to know what all this jargon means. Oz combines the salient features of object-oriented programming, by providing state, abstract data types, classes, objects and inheritance. It provides the salient features of functional programming by providing a compositional syntax, first class procedures, and lexical scoping. In fact every Oz value is first class, including procedures, threads, classes, methods, and objects. It provides the salient features of logic and constraint programming by providing logical variables, disjunctions, flexible search mechanisms and constraint programming. It is also a concurrent language where users can create dynamically any number of sequential threads that can interact with each other.

*Most of this work was done while on sabbatical at the programming Systems group at DFKI

However, in contrast to conventional concurrent languages, each Oz thread is a data-flow thread. Executing a statement in Oz proceeds only when all *real* data flow dependencies on the variables involved are resolved. Oz has its roots in a paradigm that is known as concurrent constraint programming [1].

2 Variables Declaration

We will restrict ourselves initially to the sequential programming style of Oz. You can think of Oz computations as performed by one sequential process that executes one statement after the other. We call this process a *thread*. This thread has access to a memory called the *store*. It is able to manipulate the store by reading, adding, and updating information. Information is accessed through the notion of *variables*, i.e. a thread can access information only through the variables visible to it, directly or indirectly. Oz variables are *single-assignment* variables. This notion is known from many languages including data flow, and (concurrent) logic programming languages. A single assignment variable has a number of phases in its lifetime. Initially it is introduced and later it might be assigned a value, in which case the variable becomes *bound*. Once a variable is bound it cannot itself be changed. However this does not mean that you cannot model state because a variable, as you will see later, could be bound to an object or a cell, which is statefull, i.e. the object can change its content.

A thread executing the statement

`local` $X\ Y\ Z$ **`in`** S **`end`**

will introduce three variables X, Y and Z and execute S in the scope of these variables. A variable normally starts with an upper case letter, possibly followed by an arbitrary number of alphanumeric characters. Before the execution of S the variables declared above will not have any associated values, i.e. the variables are *unbound*. Any variable in an Oz program must be introduced, except for certain pattern matching constructs to be shown later.

Another form of declaration is

`declare` $X\ Y\ Z$ **`in`** S

This is an open-ended declaration that makes X, Y and Z visible globally in S, including all statements the follows textually, unless overridden again by another variable declaration of the same variables.

3 Primary Oz Types

Oz is a dynamically-typed language. Figure 1 shows the type hierarchy of Oz. Any variable, if it ever gets a value, will be bound to a value of one of these types. Most of the types seems familiar to experienced programmers, except probably *Chunk*, *Cell*, *Space*, *FDint* and *Name*. We will discuss all of these types in due course. For the impatient reader here are some hints. The *chunck* data type allows users to introduces new abstract data types. *Cell* introduces the primitive notion of

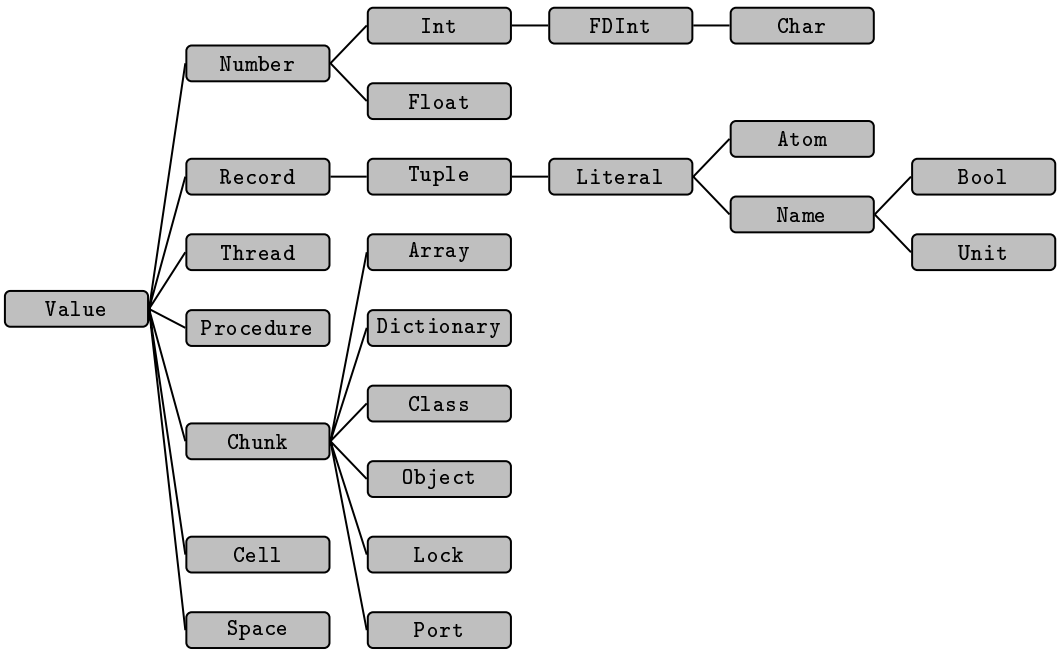

Figure 1: Oz Type Hierarchy

state-container and state modification. *Space* will be needed for advanced problem solving, and programming advanced search techniques including all you know of from an AI course. *FDint* is the type of finite domain that is used frequently in constraint programming, constraint satisfaction, and operational research. *Name* introduces anonymous unique unforgeable tokens, we also think of security !

The language is dynamically typed in the sense that when a variable is introduced its type (as well as its value) is unknown. Only when the variable is bound to an Oz value, its type becomes determined.

Hello World. Let us do like everybody else. If you are unfamiliar with Oz, here is your first Oz program. If you feed it, it will print the string `"Hello World"` in a window that appears in your screen[1]

```
{Browse "Hello World"}
```

In fact, the main reason for showing this program is to get you accustomed with one of the unconventional syntactic aspects of Oz, namely the syntax of *procedure calls/applications.*

`{Browse "Hello World"}`

is a procedure application of `Browse` on the single string argument `"Hello World"`. `Browse` is actually a pre-declared global variable that got bound to the browsing procedure when the browsing module was loaded into the system. Oz follows in

[1]What you see is an oz-application called the browser. You have to switch it to the virtual string mode to enable string visualization.

this syntactic aspect other functional languages, e.g. SCHEME and ML, with the exception of using braces instead of parentheses.

3.1 Adding Information

In Oz there are few ways of adding information to the store, or said differently of binding a variable. The most common form is using the *equality* infix operator `=`. For example:

```
X = 1
```

will bind the unbound variable `X` to the integer `1`, and add this information to the store. Now, if `X` is already assigned the value `1`, the operation is considered as performing a test on `X`. If `X` is already bound to an incompatible value (i.e. any other value different from `1`), a proper *exception*[2] will be raised.

3.2 Data Types with Structural Equality

The hierarchy starting from `Number` and `Record` in Figure 1 defines the data types of Oz whose members (values) are equal only if they are structurally similar. For example two numbers are equal if they have the same type, or one is a subtype of the other, and have the same value, e.g. both are integers and are the same number, or both are lists, and their head elements are equal as well as their respective tail lists. Structural equality allows values to be equivalent even if they are replicas occupying different (physical) memory location.

3.2.1 Numbers

The following program, introduces three variables `I F C`, and assigns `I` an integer, `F` a float, `C` the character `t` in this order, then it displays the list consisting of `I`, `F`, and `C`.

```
local I F C in
   I = 5
   F = 5.5
   C = &t
   {Browse [I F C]}
end
```

Oz support binary, octal, decimal and hexadecimal notation for integers, which can be arbitrary large. An octal starts with a leading `0`, and a hexadecimal starts with a leading `0x` or `0X`. Floats are different from integers and must have a decimal points. Other examples of floats are:

```
~3.141 4.5E3 ~12.0e~2
```

In Oz there is no automatic type conversion, so `5.0 = 5` will raise an exception. Of course there are primitive procedures for explicit type conversion. These and many others can be found in [2]. Character are subtype of integers in the range of `0 ... 255`. The standard ISO 8859-1 coding is used. Printable characters have external representation, e.g `&0` is actually the integer 48, and `&a` is `97`. Some control

[2] Exception handling is described later

characters have also a representation e.g. `&\n` is newline[3] Operations on characters, integers, and floats can be found in the library modules `Char`, `Float`, and `Int`. More generic operations on all numbers are in module `Number`.

3.2.2 Literals

Another important category of atomic types, i.e. types whose members have no internal structure, is literals. Literals are divided into atoms and names. Atoms are symbolic entities that have an identity made up of a sequence of alphanumeric characters starting with a lower case letter, or arbitrary printable characters inclosed in quotes, e.g.

```
a foo ´=´ ´:=´ ´OZ 2.0´ ´Hello World´
```

Atoms have an ordering based on lexicographic ordering. Another category of elementary entities are names. The only way to create a name is by calling the procedure `{NewName X}` where `X` is assigned a new name that is guaranteed to be world-wide unique. Names cannot be forged or printed. As will be seen later, names play an important role in the security of Oz programs. A subtype of `Name` is `Bool` which consists of two names protected from being redefined by having the reserved keywords **`true`** and **`false`**. Thus a user program cannot redefine them, and mess up all programs relying on their definition. There is also the type `Unit` which consists of the single name `unit`. This is used as a synchronization token in many concurrent programs.

```
local X Y B in
   X = foo
   {NewName Y}
   B = true
   {Browse [X Y B]}
end
```

3.2.3 Records and Tuples

Records are structured compound entities. A record has a *label* and a fixed number of components or arguments. There are also records with a variable number of arguments that are called *open records*. For now we restrict ourselves to 'closed' records. The following is a record:

```
tree(key:I value:Y left:LT right:RT)
```

It has four arguments, and the label `tree`. Each argument consists of a pair *Feature* : *Field* so the features of the above record is `key`, `value`, `left`, and `right`. The corresponding fields are the variables `I Y LT`, and `RT`. It is possible to omit the features of a record reducing it to what is known from logic programming language as a compound term. In Oz, this is called a *tuple*. So the following tuple has the same label and fields as the above record:

```
tree(I Y LT RT)
```

which is just a syntactic notation for the record:

```
tree(1:I 2:Y 3:LT 4:RT)
```

[3] All characters can be written as `\ooo`, where `o` is an octal digit.

where the features are integers starting from 1 up to the number of fields in the tuple. The following program will display a list consisting of two elements one is a record, and the other is tuple having the same label and fields:

```
declare T I Y LT RT W  in
   T = tree(key:I value:Y left:LT right:RT)
   I = seif
   Y = 43
   LT = nil
   RT = nil
   W = tree(I Y LT RT)
   {Browse [T W]}
```

The display will show:

```
[tree(key:seif value:43 left:nil right:nil)
 tree(seif 43 nil nil)]
```

Operations on records. We discuss some basic operations on records. Most operations are found in the module `Record`. To select a record component, one uses the infix operator *Record*`.`*Feature*. So

```
% Selecting a Component
   {Browse T.key}
   {Browse W.1}
```

will show `seif` twice on the display.

```
seif
seif
```

The *arity* of a record is a list of the features of the record sorted lexicographically. To display a record's arity we use the procedure `Arity`. The procedure application `{Arity R X}` will execute once R is assigned a record, and it will bind `X` to the arity of the record:

```
% Getting the Arity of a Record
   local X in {Arity T X} {Browse X} end
   local X in {Arity W X} {Browse X} end
```

will show

```
[key left right value]
[1 2 3 4]
```

Another useful operation is conditionally selecting a field of a record. The operation `CondSelect` takes a record `R`, a feature `F`, and a default field-value `D`, and a result argument `X`. If the feature `F` exists in `R`, `R.F` is assigned to `X`, otherwise the default value `D` is assigned to `X`. `CondSelect` is not really a primitive operation. It is definable in Oz. The following statements:

```
% Selecting a component conditionally
   local X in {CondSelect W key eeva X} {Browse X} end
   local X in {CondSelect T key eeva X} {Browse X} end
```

will display

```
eeva
seif
```

A common infix tuple operator used in Oz is #. So, `1#2` is a tuple of two elements, and observe that `1#2#3` is a single tuple of three elements `´#´(1 2 3)`, and not `1#(2#3)`. With the # operator there is no empty tuple notation, and single element tuple is written as `´#´(X)`.

3.2.4 Lists

As in many other symbolic programming languages, e.g Scheme and Prolog, *lists* form an important class of data structures in Oz. Lists do not belong to a single data type in Oz. They are rather a conceptual structures. A list is either the atom `nil` representing the empty list, or is a tuple using the infix operator `|` and two arguments which are respectively the head and the tail of the list. Thus a list of the first three natural numbers is represented as:

```
1|2|3|nil
```

Another convenient special notation for a *closed list*, i.e. list with determined number of elements is:

```
[1 2 3]
```

The above notation is used only for closed list, so a list whose first two elements are `a` and `b`, but whose tail is the variable `X` looks like:

```
1|2|X
```

One can also use the standard notation for lists:

```
´|´(1 ´|´(2 X))
```

Further notational variant is allowed for lists whose elements correspond to character codes. Lists written in this notation are called *strings*, e.g.

```
"OZ 2.0"
```

is the list

```
[79 90 32 50 46 48]
```

3.2.5 Virtual Strings

Virtual strings are special compound tuples which represents strings with virtual concatenation, i.e. the concatenation is performed when really needed. Virtual strings are used for I/O with files, sockets, and windows. All atoms except `nil` and # can be used, as well as numbers, strings, or #-labeled tuples with virtual strings as argument. Here is one example:

```
123#"-"#23#" is "#100
```

represents the string

```
"123-23 is 100"
```

4 Equality and the Equality Test Operator

We have so far shown simple examples of the equality statement,
i.e. `W = tree(I Y LT LR)`. These were simple enough to intuitively understand

what is going on. However, what happens when two unbound variables are equated `X = Y`, or when two large data structures are equated? Here is a short explanation.

We may think of the store as a dynamically expanding array of cells, each cell is labeled by a variable. When a variable `X` is introduced a cell is created in the store, labeled by `X`, and its value is unknown. At this point, the cell does not possess any value; it is empty as a container that might be filled later.

A variable with an empty cell is a *free* variable. The cell as a container is flexible enough to contain any arbitrary Oz value. The operation `W = tree(1:I 2:Y 3:LT 4:LR)` stores the record structure in the cell associated with `W`. Notice that we are just getting a graph structure where arcs emanating from the `W`-labeled cell are pointing to the cells of `I Y LT`, and `LR` respectively. Each arc in turn is labeled by the corresponding feature of the record. Given two variable `X` and `Y`, `X = Y` will try to *merge* their respective cells. Now we are in a position to give a reasonable account for the merge operation of `X = Y`, known as the *incremental tell* operation.

- `X` and `Y` are labels of the same cell: the operation is completed.
- `X` (`Y`) is free: merge `X` (`Y`) to the cell of `Y` (`X`), and discard the original cell of `X` (`Y`); we have a situation where a cell is labeled by a set of variables.
- `X` and `Y` are labels of different cells containing the records `Rx` and `Ry` respectively:
  - `Rx`, and `Ry` have different labels, arities, or both: the operation is completed, and an exception is raised.
  - Otherwise arguments of `Rx` and `Ry` with the same feature are pair-wise merged in arbitrary order [4].

Here are some examples of successful equality operations:

```
local X Y Z  in
    f(1:X 2:b) = f(a Y)
    f(Z a) = Z
    {Browse [X Y Z]}
end
```

will show `[a b R14=f(R14 a)]` in the browser. `R14=f(R14 a)` is the external representation of a cyclic graph.

The following example shows, what happens when variables with incompatible values are equated.

```
declare X Y Z
X = f(c a)
Y = f(Z b)
X = Y
```

The incremental tell of `X = Y` will assign `Z` the value c, but will also raise an exception that is caught by the system, when it tries to equate `a` and `b`.

[4] In general the two graphs to merge could have cycles, however any correct implementation of the merge operation will consider cell pairs for which an attempt to merge has been made as successfully merged.

Now we discuss the equality test operator ==. The basic procedure {`==` X Y R} tries to test whether X and Y are equal or not, and returns the result in R. It returns **true** if the graphs starting from the cells of X and Y have the same structure, with each pair-wise corresponding cells having identical Oz values or are the same cell. It returns **false** if the graphs have different structure, or some pair-wise corresponding cells have different values. It suspends when it arrives at pair-wise corresponding cells that are different, but at least one them is free. Now remember if a procedure suspends, the whole thread suspends ! This does not seem useful however, as you will see, it becomes a very useful operation when multiple thread start interacting with each other.

The equality test is normally used as a functional expression, rather than a statement. As shown is the following example:

```
% See, lists are just tuples, which are just records

local L1 L2 L3 Head Tail in
   L1 = Head|Tail
   Head = 1
   Tail = 2|nil

   L2 = [1 2]
   {Browse L1==L2}

   L3 = ´|´(1:1 2:´|´(2 nil))
   {Browse L1==L3}
end
```

5 Basic Control Structures

We have already seen basic statements in Oz. Introducing new variables and sequencing of statements:

$$S_1 \ S_2$$

Reiterating again, a thread executes statements in a sequential order. However a thread, contrary to conventional languages, may suspend in some statement, so above a thread has to complete execution of S_1, before starting S_2[5].

Skip. The statement `skip` is the empty statement.

5.1 Conditionals

Simple Conditional. A statement having the following form:

$$\textbf{if } X_1 \ \ldots \ X_n \textbf{ in } C \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}$$

where C is a sequence of equalities, is called a simple conditional. The subexpression

$$\textbf{if } X_1 \ \ldots \ X_n \textbf{ in } C \textbf{ then } S_1$$

[5] S_2 may not be executed at all, if an exception is raised in S_1.

is usually called a *clause*, and

$$X_1 \ \ldots \ X_n \ \textbf{in} \ C$$

is the *condition* of the clause. A thread executing such as conditional will first check whether the condition: "There are $X_1 \ldots X_n$ such that C is true" is satisfied or falsified by the current state of the store. If the condition is satisfied statement S_1 is executed; if it is falsified the thread executes S_2; and if neither the thread suspends.

The most common types of conditions are simple equalities as:

```
X = true           % X is bound to true in the store
Y Z in X = f(Y Z) % X is bound to a tuple of 2 arguments
X = Y         % X and Y label the same cell in the store
```

Comparison Procedures. Oz provides a number of built-in tertiary procedures used for comparison. These include `==` that we have seen earlier as well as `\=`, `=<`, `<`, `>=`, `>`, **`andthen`**, and **`orelse`**. Common to these procedures is that they are used as boolean functions in an infix notation.

The following example illustrates the use of a conditional in conjunction with the greater-than operator `>`. In this example `Z` is bound to the maximum of `X` and `Y`, i.e. to `Y`:

```
local X Y F Z in
   X = 5
   Y = 10
   F = X > Y
   if F = true then
      Z = X
   else
      Z = Y
   end
end
```

Parallel Conditional. A parallel conditional is of the from:

$$\begin{array}{c}\textbf{if} \ C_1 \ \textbf{then} \ S_1 \\ \texttt{[]} \ C_2 \ \textbf{then} \ S_2 \\ \vdots \\ \textbf{else} \ S_n \ \textbf{end}\end{array}$$

A parallel conditional is executed by evaluating all conditions C_1 C_2 ... in an *arbitrary* order, possibly concurrently. If one of the conditions, say C_i is true, its corresponding statement S_i is chosen. If all conditions are false, the else statement S_n is chosen, otherwise the executing thread suspends. Parallel conditionals are useful mostly in concurrent programming, e.g. for programming time-out on certain events. However it is often used in a deterministic manner with mutually exclusive conditions. So, the above example could be written as:

```
local X Y F Z in
   X = 5
   Y = 10
```

```
local X Y Z in
   X = 5
   Y = 10
   case X >= Y then Z = X
   else Z = Y end
end
```

Figure 2: Using a Case Statement

```
   F = X >= Y
   if F = true then
      Z = X
   [] F = false then
      Z = Y
   end
end
```

Notice that the **else** part is missing. This is just a short-hand notation for an **else** clause that raises an exception.

Case Statement. Oz provides an alternative conditional syntax that encourages the use of a very simple form of conditionals. This is called the *case* statement. The following is the simplest form:

$$\textbf{case}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}$$

where B is a variable that should be bound to a boolean value. if $B =$ **true** S_1 is executed, otherwise if $B =$ **false** S_2 is executed. This is equivalent to:

$$\textbf{if}\ B\ =\ \textbf{true then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}$$

Our example using if-statement could be now written as shown in Figure 2

Since a case-statement is defined in terms of an if-statement, it is merely a notational convenience.

5.2 Procedural Abstraction

Procedure Definition. Procedure abstraction is a primary abstraction in Oz. A procedure can be defined, passed around as argument to another procedure, or stored in a record. A procedure definition is a statement that has the following structure.

$$\textbf{proc}\ \{P\ X_1\ \ldots\ X_n\}\ S\ \textbf{end}$$

Assume that the variable P is already introduced, executing the above statement will create a procedure, consisting of a unique name a and an abstraction:
$\lambda(X_1 \ldots X_n) \cdot S$. This pair is stored in the cell labeled by P.

A procedure in Oz has a unique identity, given by its name, and is distinct from all other procedures. Two procedure definitions are always different, even if they look similar. Procedures are the first Oz values that we encounter, whose equality is based on name or *token equality*, others include objects classes, and chunks.

On Lexical Scoping. In general the statement S in a procedure definition will have many 'syntactic' variable occurrences. Some occurrences are *syntactically bound* and others are *free*. A variable occurrence X in S is bound if it is in the scope of the procedure formal-parameter X, or in the scope of a variable introduction statement that introduces X[6]. Otherwise the variable occurrence is free. Each variable occurrence in an Oz is eventually bound by the closest statically surrounding variable-binding construct.

We have already seen how to apply procedures. Let us now show our first procedure definition. In Figure 2 we have seen how to compute the maximum of two numbers or literals. We abstract this code into a procedure.

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

Anonymous Procedures and Variable Initialization. One could ask why a variable gets bound to a procedure in a way different from the normal way where an equality is used: `X = Value`. The answer is that what you have seen is just a syntactic variant to the equivalent form `P = proc {$ X Y ...} S end` where the R.H.S. defines an anonymous procedural value. This is equivalent to `proc {P X Y ...} S end`.

In Oz we can initialize a variable immediately while it is being introduced by using an equality between **local** and **in**, in the statement **local ... in ... end**.

So the previous example could be written as follows, where we also use anonymous procedures.

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

Now let us understand variable initialization in more detail. The general rule says that, in a variable initialization equality, only the variables occurring on the L.H.S of the equality are being defined. Consider the following example:

[6] This rule is approximate. Methods also bind variable occurrences as well as patterns.

```
local
   Y = 1
in
   local
      M = f(M Y)
      [X1 Y] = L
      L = [1 2]
   in
   {Browse [M L]}
   end
end
```

First `Y` is introduced and initialized in the outer **`local ... in ... end`**. Then in the inner **`local ... in ... end`** all variable on the L.H.S are introduced, i.e. `M Y X1` and `L`. So the outer `Y` is invisible in the innermost **`local ... end`** statement. The above statement is equivalent to:

```
local Y in
   Y = 1
   local M X1 Y L in
      M = f(M Y)
      L = [X1 Y]
      L = [1 2]
      {Browse [M L]}
   end
end
```

If we want `Y` to denote the variable in the outer scope, we have to suppress the introduction of the inner `Y` in the L.H.S. by using the exclamation mark `!` as follows. `!` is only meaningful in the L.H.S. of an equality.

```
local
   Y = 1
in
   local
      M = f(M Y)
      [X1 !Y] = L
      L = [1 2]
   in
   {Browse [M L]}
   end
end
```

5.3 Pattern Matching

Let us consider a very simple example: insertion of elements in a binary tree. A binary tree is either empty, represented by `nil`, or is a tuple of the form `tree(Key Value TreeL TreeR)`, where `Key` is a key of the node with the corresponding value `Value`, and `TreeL` is the left subtree having keys less than `Key`, and `TreeR` is the right subtree having keys greater than `key`. The procedure `Insert` takes four arguments, three of them are input arguments `Key`, `Value` and `TreeIn`, and one output argument `TreeOut` to be bound to the resulting tree after insertion. The program is shown in Figure 3. The symbol ? before `TreeOut` is a voluntary documentation

```
declare
proc {Insert Key Value TreeIn ?TreeOut}
   if TreeIn = nil then TreeOut = tree(Key Value nil nil)
   [] K1 V1 T1 T2 in TreeIn = tree(K1 V1 T1 T2) then
      case Key == K1 then TreeOut = tree(Key Value T1 T2)
      elsecase Key < K1 then
         local T in
            TreeOut = tree(K1 V1 T T2)
            {Insert Key Value T1 T}
         end
      else
         local T in
            TreeOut = tree(K1 V1 T1 T)
            {Insert Key Value T2 T}
         end
      end
   end
end
```

Figure 3: Insertion in a Binary Tree

to denote that the argument plays the role of an output argument. The procedure works by cases as obvious. First depending on whether the tree is empty or not, and in the latter case depending on a comparison between the key of the node encountered and the input key.

Notice the use of **`case ... then ... elsecase ... else ... end`** with the obvious meaning.

Pattern Matching. In Figure 3 it is tedious to introduce the local variables in the clause `[] K1 V1 T1 T2` **`in`** `TreeIn = tree(K1 V1 T1 T2)` **`then`** `...` Oz provides a pattern matching case statement, that allows implicit introduction of variables in the patterns. Two forms exist for a pattern-matching case-statement:

`case` X
`of` $Pattern_1$ **`then`** S_1
`elseof` $Pattern_2$ **`then`** S_2
`elseof` `...`
`else` S
`end`

and

`case` X
`of` $Pattern_1$ **`then`** S_1
`[]` $Pattern_2$ **`then`** S_2
`[] ...`
`else` S **`end`**

All variables introduced in a $Pattern_i$ are implicitly declared, and have a scope stretching over the corresponding clause. In the first case-matching statement, the patterns are tested in order. In the second, called *parallel* case-matching statement,

```
% case for pattern matching
declare
proc {Insert Key Value TreeIn ?TreeOut}
   case TreeIn
   of nil then TreeOut = tree(Key Value nil nil)
   [] tree(K1 V1 T1 T2) then
      case Key == K1 then TreeOut = tree(Key Value T1 T2)
      elsecase Key < K1 then
         T in
            TreeOut = tree(K1 V1 T T2)
            {Insert Key Value T1 T}
      else
         T in
            TreeOut = tree(K1 V1 T1 T)
            {Insert Key Value T2 T}
      end
   end
end
```

Figure 4: Insertion in a Binary Tree

the order is indeterminate. The else part may be omitted, in which case an exception is raised if all matches fail. Again, in each pattern one may suppress the introduction of new local variable by using `!`. For example in

```
case f(X1 X2) of f(!Y Z) then ... else ... end
```

matching of `X1` is against the value of the external variable `Y`. Now remember again that the case statement and its executing thread may suspend if current value of `X1` is insufficient to decide the result of the matching.

Have all this said, Figure 4 shows the tree-insertion procedure using a matching case-statement. We have also reduced the syntactic nesting by abbreviating:

```
local T in
    TreeOut = tree( ... T ... )
    {Insert ... T}
end
```

into:

```
T in
    TreeOut = tree( ... T ... )
    {Insert ... T}
```

The expression we may match against, could be any record structure, and not only a variable. This allows multiple argument-matching, as shown in Figure 5, which depicts a classical stream-merge procedure. Here the use of parallel-case is essential.

```
proc {Merge Xs Ys ?Zs}
   case Xs#Ys
   of nil#Ys then Zs = Ys
   [] Xs#nil then Zs = Xs
   [] (X|Xr)#Ys then
        Zr in
        Zs = X|Zr
        {Merge Ys Xr Zr}
   [] Xs#(Y|Yr) then
        Zr in
        Zs = Y|Zr
        {Merge Yr Xs Zr}
   end
end
```

Figure 5: Binary Merge of Two Lists

5.4 Nesting

Let us use our `Insert` procedure as defined in Fig. 4. The following statement inserts a few nodes in an initially empty tree. Note that we had to introduce a number of intermediate variables to perform our sequence of procedure calls.

```
local T0 T1 T2 T3 in
   {Insert seif 43 nil T0}
   {Insert eeva 45 T0 T1}
   {Insert rebecca 20 T1 T2}
   {Insert alex 17 T2 T3}
   {Browse T3}
end
```

Oz provides syntactic support for nesting one procedure call inside another statement at an expression position. So in general:

```
local Y in
   {P ... Y ...}
   {Q Y ... }
end
```

could be written as:

```
   {Q {P ... $ ...} ... }
```

`$` is called a nesting marker, and thus the variable `Y` is eliminated. The rule, to revert to the flattened syntax, is that a nested procedure call, inside a procedure call, is moved *before* the current statement, and a new variable is introduced with one occurrence replacing the nested procedure call, and the other occurrence replacing the nesting marker.

Functional Nesting. Another form of nesting is called functional nesting: a procedure `{P X ... R}` could be considered as a function, its result is the argument

```
proc {Merge Xs Ys ?Zs}
   case Xs#Ys
   of nil#Ys then Zs = Ys
   [] Xs#nil then Zs = Xs
   [] (X|Xr)#Ys then Zs = X|{Merge Ys Xr $}
   [] Xs#(Y|Yr) then Zs = Y|{Merge Yr Xs $}
   end
end
```

Figure 6: Binary Merge of Two Lists in Nested Form

R, and therefore {P X ...} could considered as a function call that can be inserted in any expression instead of the result argument R. So

```
{Q {P X ... } ... }
```

is equivalent to:

```
local R in
   {P X ...  R}
   {Q R ... }
end
```

Now back to our example, a more concise form using functional nesting is:

```
{Browse {Insert alex 17
                {Insert rebecca 20
                        {Insert eeva 45 {Insert seif 43 nil}}}}}
```

There is one more rule to remember. It has to do with a nested application inside a record or a tuple as in:

```
Zs = X|{Merge Xr Ys $}
```

Here, the nested application goes *after* the record construction statement; Do you know why ?. So we have

```
local Zr in
   Zs = X|Zr
   {Merge Xr Ys Zr}
end
```

We can now rewrite our Merge procedure as shown in Figure 6, where we use nested application.

5.5 Procedures as Values

Since we have been inserting elements in binary trees, let us define a program that checks if is data structure is actually a binary tree. The procedure BinaryTree shown in FigureBinaryTree checks a structure to verify whether it is a binary tree or not, and accordingly returns **true** or **false** in its result argument B. Notice that we also defined the auxiliary local procedure And.

Consider the call

```
{And {BinaryTree T1} {BinaryTree T2} B}
```

```
% What is a binary tree ?
declare BinaryTree
local
   proc {And B1 B2 B}
      case B1 then
         case B2 then B = true else B = false end
      else B = false end
   end
in
   proc {BinaryTree T B}
      case T
      of nil then B = true
      [] tree(K V T1 T2) then
         {And {BinaryTree T1} {BinaryTree T2} B}
      else B = false end
   end
end
```

Figure 7: Checking a Binary Tree

It is certainly doing unnecessary work: according to our nesting rules it evaluates its second argument even if the first is false. One can fix this problem by making an new procedure `AndThen` that takes as its first two arguments two procedure values, and calls the second argument only if the first returns **`false`**. Thus getting the effect of delaying the evaluation of its arguments until really needed. The procedure is shown in Figure 9. `AndThen` is the first example of a higher-order procedure, i.e. a procedure that takes other procedures as argument, and may return other procedures as results. In our case `AndThen` returns just a Boolean value, but in general we are going to see other examples where procedures return procedures as result. As in functional languages, higher order procedures are invaluable abstraction devices that helps creating generic reusable abstractions.

Control Abstractions. Higher-order procedures are used in Oz to define various control abstractions. In modules `Control` and `List` as well as many others you will find many control abstractions. Here are some examples.

The procedure `{For From To Step P}` is an iterator abstraction that applies the unary procedure `P` (normally saying the procedure `P/1` instead) to integers from `From` to `To` proceeding in steps `Step`. Notice that use of the empty statement `skip`. Executing `{For 1 10 1 Browse}` will display the integers `1 2 ... 10`

Another control abstraction that is often used, is the `ForAll` iterator defined in the `List` module. `ForAll` applies a unary procedure on all the elements of a list, in the order defined by the list. Think what happens if the list is produced inclemently by another concurrent thread.

```
declare
proc {ForAll Xs P}
   case Xs
   of nil then skip
```

```
declare BinaryTree
local
   proc {AndThen BP1 BP2 B}
      case {BP1} then
         case {BP2} then B = true else B = false end
      else B = false end
   end
in
   proc {BinaryTree T B}
      case T
      of nil then B = true
      [] tree(K V T1 T2) then
         {AndThen proc {$ B1}{BinaryTree T1 B1} end
                  proc {$ B2}{BinaryTree T2 B2} end
                  B}
      else B = false end
   end
end
```

Figure 8: Checking a Binary Tree lazily

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

Figure 9: the For Iterator

```
   [] X|Xr then
      {P X}
      {ForAll Xr P}
   end
end
```

6 Functional Notation

Oz provides functional notation as syntactic convenience. We have seen that a procedure call:

{P X_1 ... X_n R}

could be used in a nested expression as a function call:

{P X_1 ... X_n}

Oz also allows functional abstractions directly as syntactic notation for procedures. So the following function definition:

fun {F $X_1 \ldots$ X_n} S E **end**

where S is a statement and E is an expression corresponds to the following procedure definition:

proc {F $X_1 \ldots$ X_n R} S $R = E$ **end**

The exact syntax for functions as well as their unfolding into procedure definitions is defined in [3]. Here we rely on the reader's intuition. Roughly speaking, the general rule for syntax formation of functions looks very similar to how procedures are formed, with the exception that whenever a thread of control, in a procedure, ends in a statement the corresponding function ends in an expression.

The program shown in Figure 10 is the functional equivalent to the program shown in Figure 8. Notice how the function `AndThen` is unfolded into the procedure `AndThen`. Here are a number of steps that give some intuition of the transformation process. All the intermediate forms are legal Oz programs

```
fun {AndThen BP1 BP2}
   case {BP1} then
      case {BP2} then true else false end
   else false end
end
```

Make a procedure by introducing a result variable `B`

```
proc {AndThen BP1 BP2 B}
   B = case {BP1} then
            case {BP2} then true else false end
       else false end
end
```

Move the result variable into the outer *case-expression* to make it a *case-statement*:

```
proc {AndThen BP1 BP2 B}
   case {BP1} then
       B = case {BP2} then true else false end
   else B = false end
end
```

```
% Syntax Convenience: functional notation
local
   fun {AndThen BP1 BP2}
      case {BP1} then
         case {BP2} then true else false end
      else false end
   end
   fun {BinaryTree T}
      case T
      of nil then true
      [] tree(K V T1 T2) then
         {AndThen fun {$}{BinaryTree T1} end
                  fun {$}{BinaryTree T2} end}
      else false end
   end
end
```

Figure 10: Checking a Binary Tree lazily

Move the result variable into the inner *case-expression* to make it a *case-statement*, and we are done:

```
   proc {AndThen BP1 BP2 B}
      case {BP1} then
          case {BP2} then B = true else B = false end
      else B = false end
   end
```

If you are a functional programmer, you can cheer up! You have your functions, including higher-order ones, and similar to lazy functional languages Oz allows certain forms of tail (recursion) optimizations that are not found in certain (strict[7]) functional languages including Standard ML, SCHEME and the concurrent language Erlang. However functions in Oz are not lazy. This property is easily programmed, using the concurrency constructs and the variables. Here is an example of the well known higher order function `Map`. It is tail recursive in Oz but not in Standard ML nor in SCHEME.

```
declare
fun {Map Xs F}
   case Xs
   of nil then nil
   [] X|Xr then {F X}|{Map Xr F}
   end
end
{Browse {Map [1 2 3 4] fun {$ X} X*X end}}
```

andthen and orelse. After all we have been doing a lot of work for nothing! Oz already provides the Boolean lazy (non-strict) version of the Boolean function

[7] A strict function evaluates all its argument first before executing the function.

```
fun {BinaryTree T}
    case T of nil then true
    [] tree(K V T1 T2) then
       {BinaryTree T1} andthen {BinaryTree T2}
    else false end
end
```

Figure 11: Checking a Binary Tree lazily

`And` and `Or` as the Boolean operators **`andthen`** and **`orelse`** respectively. The former behaves like the function `AndThen`, and the latter evaluates its second argument only if the first argument evaluates to **`false`**. As usual these operators are not primitives, they are defined in Oz. Figure 11 defines the final version of `BinaryTree`.

To Function or not to Function. The question is when to use functional notation, and when not. The honest answer is that it is up to you! I will tell you my personal opinion. Here are some rules of thumb:

- First what I do not like. Given that you defined a procedure `P` do not call it as a function, i.e. do not use functional nesting for procedures. Use instead procedural nesting, with nesting marker, as in the `Merge` example. And given that you defined a function, call it as function.
- I tend to use function definitions when things are really functional, i.e. there is one output and, possibly many inputs, and the output is a mathematical function of the input arguments.
- I tend to use procedures in most of the other cases, i.e. multiple outputs or nonfunctional definition due to stateful data types or nondeterministic definitions[8].
- One may relax the previous rule and use functions when there is a clear direction of information-flow although the definition is not strictly functional. Hence by this rule we can write the binary-merge in Figure 6 as a function although it is definitely not. After all functions are concise.

7 Modules and Interfaces

Modules, also known as packages, are collection of procedures and other values[9] that are constructed together to provide certain related functionality. A module typically has a number of private procedures that are not visible outside the module and a number of interface procedures that provides the external services of the module. The interface procedures provide the interface of the module. In Oz there are no syntactic support for modules or interfaces. Instead the lexical scoping of

[8] In fact I use mostly objects except for typical constraint and logic programming applications

[9] Classes, objects, etc.

the language and the record data-type suffices to construct modules. The general construction looks as follows. Assume that we would like to construct a module called `List` that provides a number of interface procedures for appending, sorting and testing membership of lists. This would look as follows.

```
declare List in
local
   proc {Append ... }  ... end
   proc {Sort ... } ... {MergeSort ...} ... end
   proc {Member ...} ... end
   proc {MergeSort ...} ... end
in
   List = list(append: Append
               sort: Sort
               member: Member
               ... )
end
```

To access `Append` procedure outside the module List by `List.append`. Notice that the procedure `MergeSort` is private to the list module. Oz standard modules [2] follows the above structure. Often some of the procedures in the a module are made global without the module prefix. The are several ways to do that. Here is one way to make `Append` Externally visible.

```
declare List Append in
Append = List.append
local
   proc {Append ... }  ... end
   ...
in
   List = list(append: Append
               sort: Sort
               member: Member
               ... )
end
```

Modules are often stored on files, for example the `List` module could be stored on the file list.oz. This file could be inserted into another file by using the directive `\insert ´list.oz´`. Consult the user manual [4] for details.

Modules in Oz could be nested. This is apparent from compositional syntax of Oz.

8 Concurrency

So far we seen only one thread executing. It is time to introduce concurrency. In Oz an new concurrent thread of control is spawned by:

thread S **end**

Executing this statement, a thread is forked that runs concurrently with the current thread. The current thread resumes immediately with the next statement. Each nonterminating thread, that is not blocking, will be eventually allocated a time slice of the processor. This means that threads are executed fairly. However, there are

```
declare
fun {Map Xs F}
   case Xs
   of nil then nil
   [] X|Xr then thread {F X} end |{Map Xr F}
   end
end
```

Figure 12: A Concurrent Map function

three priority levels: *high*, *medium* and *low* that determines how often a runnable thread is allocated a time slice. So a high priority thread cannot starve a low priority one. Priority determines only how large piece of the processor cake a thread can get.

One can also get a reference to a new thread by asking for a reference when a new thread is spawned as follows:

thread X **runs** S **end**

Having a reference to a thread enables operations on threads such as terminating it, or raising an exception in it. Thread operations are defined module `Thread` [2].

Let us see what we can do with threads. First remember that each thread is a data flow thread that blocks on data dependency. Consider the following program:

```
declare
X0 X1 X2 X3
thread
   local Y0 Y1 Y2 Y3 in
      {Browse [Y0 Y1 Y2 Y3]}
      Y0 = X0+1
      Y1 = X1+Y0
      Y2 = X2+Y1
      Y3 = X3+Y2
      {Browse completed}
   end
end
{Browse [X0 X1 X2 X3]}
```

If you input this program and watch the display of the Browser tool, the variables will appear unbound. If you now input the following statements one at a time:

```
X0 = 0
X1 = 1
X2 = 2
X3 = 3
```

you will see how the thread resumes and then suspends again. First when `X0` is bound the thread can execute `Y0 = X0+1` and suspends again because it needs the value of `X1` while executing `Y1 = X1+Y0`, and so on.

The program shown in Figure 12 defines a concurrent `Map` function. Notice that **`thread ... end`** is used here as an expression. Let us discuss the behavior of this program. If we enter the following statements:

```
declare
fun {Fib X}
   case X
   of 0 then 1
   [] 1 then 1
   else thread {Fib X-1} end  + {Fib X-2} end
end
```

Figure 13: A Concurrent Fibonacci function

```
declare
F X Y Z
{Browse thread {Map X F} end}
```

a thread executing `Map` is created, which suspends immediately in the case statement because `X` is unbound. If then enter the following statements:

```
X = 1|2|Y
fun {F X} X*X end
```

the main thread will traverse the list creating two threads for the first two arguments of the list, **thread** `{F 1}` **end** and **thread** `{F 2}` **end**, and then suspends again on the tail of the list `Y`. Finally

```
Y = 3|Z
Z = nil
```

will complete the computation of the main thread and the newly created thread **thread** `{F 3}` **end**, resulting in the final list `[1 4 9]`.

The program shown in Figure 13 is a concurrent divide and conquer program, that is very inefficient way to compute the *Fibonacci* function. This program creates an exponential number of threads! See how it is easy to create concurrent threads. You may use this program to test how many threads your Oz installation can create. Try `{Fib 24}`, and use the panel program in your Oz menu. If it works try a larger number.

The whole idea of explicit thread creation in Oz is to enable the programmer to structure his/her application in a modular way. Therefore create threads only when the application need it, and not because concurrency is fun.

Time. In module `Time` we can find a number of useful real-time procedures. Among them are `{Alarm I ?U}` which creates immediately its own thread, and binds `U` to `unit` after `I` milliseconds. `{Delay I}` suspends the executing thread for, a least, `I` milliseconds and then reduces to `skip`.

The program shown in Figure 15 starts two threads, one displays `Ping` periodically after 500 milliseconds, and the other `Pong` after 600 milliseconds. Some `Ping`s will be displayed immediately after each other because of the periodicity difference.

Stream Communication. The data-flow property of Oz easily enables writing threads that communicate through streams in a producer-consumer pattern. A

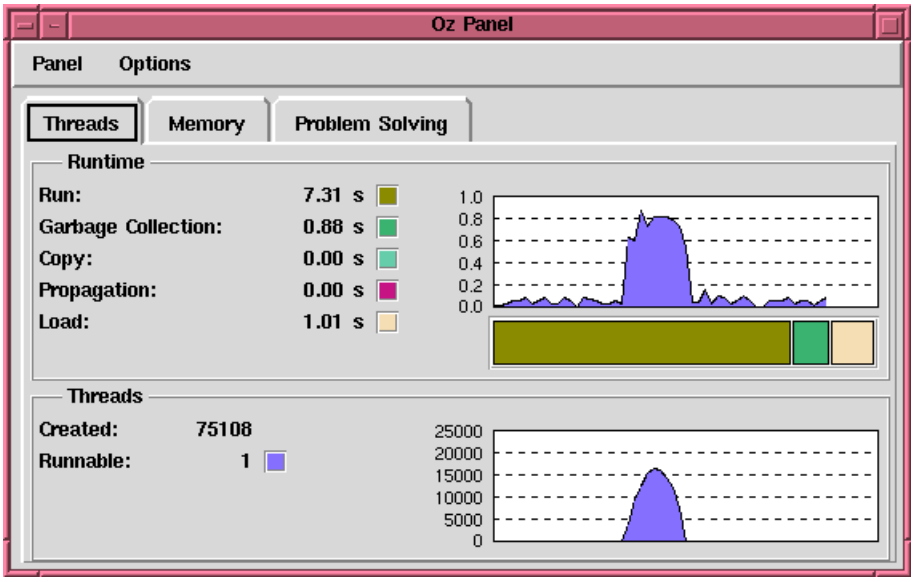

Figure 14: The Oz Panel Showing Thread Creation in `{Fib 26 X}`

```
local
   proc {Ping N}
      case N==0 then {Browse ´ping terminated´}
      else {Delay 500} {Browse ping} {Ping N-1} end
   end
   proc {Pong N}
      {For 1 N 1
        proc {$ I} {Delay 600} {Browse pong} end }
      {Browse ´pong terminated´}
   end
in
   {Browse ´game started´}
   thread {Ping 50} end
   thread {Pong 50} end
end
```

Figure 15: A Ping Pong Program

```
declare
fun {Generator N}
   case N > 0 then N|{Generator N-1}
   else nil end
end
fun {Sum L}
   fun {Sum1 L A}
      case L
      of nil then A
      [] X|Xs then {Sum1 Xs A+X}
      end
   end
in {Sum1 L 0}
end
```

Figure 16: Summing the Elements in a List

stream is a list that is created incremently by one thread (the producer) and subsequently consumed by one or more threads. The threads consume the same elements of the stream. For example, the program in Figure 16 is an example of stream communication, where the producer generates a list of numbers and the consumer sums all the numbers.

Try the program in Figure 16 by running the following program:

```
local L in
   thread L = {Generator 150000} end
   thread {Browse {Sum L}} end
```

It should produce the number `11250075000`. Let us understand the working of stream communication. A producer incremently create a stream (a list) as in the following where it is producing a `volvo`. This happens in general in an eager fashion.

```
   fun {Producer ...} ... volvo|{Producer ...} ... end
```

The consumer waits on the stream until items arrives, then it is consumed as in:

```
   proc {Consumer Ls ...}
      case Ls of volvo|Lr then Consume volvo ... end
      {Consumer Lr}
   end
```

The data flow behavior of the *case − statement* lets the consumer wait until the arrival of the next item of the stream. The recursive call allows the consumer to iterate the action over again. The following pattern avoids the use of recursion by using an iterator instead:

```
   proc {Consumer Ls ...}
      {ForAll Ls
         proc {$ Item}
            case Item of volvo then Consume volvo ... end
         end }
   end
```

```
declare
fun {Producer N}
   case N > 0 then
      volvo|{Producer N-1}
   else then nil end
end
proc {Consumer Ls}
   proc {Consumer Ls N}
      case Ls
      of nil then skip
      [] volvo|Lr then
         case N mod 1000 == 0 then
            {Browse ´riding a new volvo´}
         else skip end
         {Consumer Lr N+1}
      else {Consumer Lr N} end
   end
in {Consumer Ls 1}
end
```

Figure 17: Producing volvo's

Figure 17 shows a simple example using this pattern. The consumer counts the cars received. Each time it receives 1000 car it prints a message on the display of the Browser. You may run this program using:

```
{Consumer thread {Producer 10000} end}
```

Notice that the consumer was written using the *recursive* pattern. Can we write this program using the iterative `ForAll` construct? This is not possible because the consumer carries an extra argument N that accumulates a result which is passed to the next recursive call. The argument corresponds to some kind of *state*. In general there are two solutions, either introduce the concept of stateful data structures, which we will do in Section ??, or define another iterator that passes the state around. In our there case there are some iterators that fit our needs in the module `List` in [2]. First we need an iterator that filter away all items except volvos. We can use `{Filter +Xs +P Ys}` which outputs in `Ys` all elements that satisfies `P/2` as a Boolean function. The second is `{List.forAllInd +Xs +P}` which is similar to `ForAll`, but `P/2` is a binary procedure that takes the index of the current element of the list, , starting from `1`, as its first argument, and the element as its second argument. Here is the program:

Thread Priority and Real Time. Try to run the program using the following statement:

```
{Consumer thread {Producer 5000000}}
```

Switch on the panel and observe the memory behavior of the program. You will quickly notice that this program does not behave well. The reason have to do with the asynchronous message-passing. If the producer sends messages, i.e. create

```
proc {Consumer Ls}
   fun {IsVolvo X} X == volvo end
   Ls1
in
   thread Ls1 = {Filter Ls IsVolvo} end
   {List.forAllInd Ls1
      proc {$ N X}
         case N mod 1000 == 0 then
            {Browse ´riding a new volvo´}
         else skip end
      end }
end
```

new elements in the stream, in a faster rate than the consumer can consume, more and more buffering will be needed until the system starts to break down[10]. There are a number of ways to solve this problem, one is to create a bounded buffer between producers and consumers which we will be discussed later. Another way is to experimentally control the rate of thread execution so that the consumers gets a larger time-slice than the producers. The modules `Thread`, in [2], and `System`, in [4], provide a number of operations pertinent to threads. Some of these are summarized in Table 1.

Procedure	Description
`{Thread.state +T A}`	returns current state of `T`
`{Thread.suspend +T}`	suspends `T`
`{Thread.resume +T}`	resumes `T`
`{Thread.terminate +T}`	terminates `T`
`{Thread.injectException +T +E}`	raises exception `E` in `T`
`{Thread.this T}`	returns the current thread `T`
`{Thread.setPriority +T +P}`	sets `T`'s priority
`{Thread.setThisPriority +P}`	sets current thread's priority
`{System.get priorities P}`	gets system-priority ratios
`{System.set priorities(high:+X medium:+Y)}`	sets system-priority ratios

Table 1: Thread Operations

The system procedure `{System.set priorities(high:+X medium:+Y)}` sets the processor-time ratio to `X:1` between high-priority threads and medium-priority thread. It also sets the processor-time ratio to `Y:1` between medium-priority threads and low-priority thread. `X` and `Y` are integers. So if we execute

```
{System.set priorities(high:+10 medium:+10)}
```

each for `10` time-slices allocated to runnable high-priority thread, the system will allocate one time-slice for medium-priority threads, and similarly between medium and low priority threads. Within the same priority level scheduling is fair and round-robin.

[10] Ironically in the research project Perdio, developing an Internet-wide distributed Oz system, stream communication across machines works better because of designed flow-control mechanisms that suspend producers with the network buffers are full.

Now let us make our producer-consumer program work. We give the producer low priority, and the consumer high. We also set the priority ratios to `10:1` and `10:1`.

```
   L
in
   {System.set priorities(high:+10 medium:+10)}
   thread
      X in
      {Thread.setThisPriority low}
      L = {Producer 5000000}
   end
   thread
   {Thread.setThisPriority high}
   {Consumer L}
   end
end
```

Demand-driven Execution. An extreme alternative solution is to make the producer lazy. Only producing an item when the consumer requests one. A consumer, in the case, construct the stream with empty boxes (unbound variables). The producer waits for the empty boxes (unbound variables) appearing of the stream. It then fills the boxes (binds the variables). The general pattern of the producer is a follows.

```
proc {Producer Xs}
   case Xs of X|Xr then I in Produce I X=I ... {Producer Xr} end
end
```

The general pattern of the consumer is as follows.

```
proc {Consumer ... Xs}
   X Xr in ... Xs = X|Xr Consume X ... {Consumer ... Xr}
end
```

The program shown in Figure 18 is a demand driven version of the program in Figure 17. You can run with very large number of Volvo's!

Thread Termination-Detection. We have seen how thread are forked using the statement **thread** `S` **end**. A natural question that arises is how to join back back into the original thread of control. In fact this is a special case of detecting termination of multiple threads, and making another thread wait on that event. The general scheme is quite easy, since Oz is a data-flow language.

```
   thread T1  X1 =unit end
   thread T2  X2 = X1   end
          ...
   thread Tn  Y = Xn-1 end
   {Wait Y}
   MainThread
```

When All threads terminate the variables $X_1 \ldots Y$ will be merged together labeling a single box that contains the value `unit`. `{Wait X}` suspends the current thread until `X` is bound. In Figure 19 refines a higher-order construct (combinator), that

```
local
   proc {Producer L}
      case L of X|Xs then X = volvo {Producer Xs}
      [] nil then {Browse ´end of line´}
      end
   end
   proc {Consumer N L}
      case N==0 then L = nil
      else
         X|Xs = L
      in
         case X of volvo then
            case N mod 1000 == 0 then
               {Browse ´riding a new volvo´}
            else skip end
            {Consumer N-1 Xs}
         else {Consumer N Xs} end
      end
   end
in
   {Consumer 10000000 thread {Producer $} end}
end
```

Figure 18: Producing volvo's

implements a concurrent composition control construct. It takes a single argument that is a list of nullary procedures. When it is executed the procedures are forked concurrently. The next statement is executed only when all procedures in the list terminate.

9 Stateful Data Types

Oz provides set of stateful data types. These include ports, objects, arrays, and dictionaries (hash tables). These data types are abstract in the sense that they are characterized only by the set of operations performed on the members of the type. Their implementation is always hidden, and in fact different implementations exist but their corresponding behavior remains the same[11]. Each member is always unique by conceptually tagging it with an Oz-name upon creation. A member is created by an explicit creation operation. A type test operation always exist. And a member ceases to exist when it is no longer accessible.

9.1 Ports

Ports is such an abstract data-type. A Port `P` is an asynchronous communication channel that can be shared among several sender. A port has a stream associated

[11] For examples objects are implemented in a totally different way depending on the optimization level of the compiler.

```
declare
proc {Conc Ps}
   proc {Conc Ps I O}
      case Ps
      of P|Pr then
         M in
         thread {P} M = I end
         {Conc Pr M O}
      [] nil then O = I
      end
   end
in
   {Wait {Conc Ps unit $}}
end
```

Figure 19: Concurrent Composition

with it. The operation: `{NewPort S P}` creates a port `P` and initially connect to the variable `S` taking the role of a stream. The operation: `{Send +P M}` will append the message `M` to the end of the stream. The port keeps track of the end of the stream as its next insertion point. The operation `{IsPort +P B}` checks whether `P` is a port. The following program shows a simple example using ports:

```
declare S P
P = {Port.new S}
{Browse S}
{Send P 1}
{Send P 2}
```

If you enter the above statements incremently you will observe that S gets incrementally more defined

```
S
1|_
1|2|_
```

Ports are more expressive abstractions than pure stream communication discussed in Section 8, since they can shared among multiple thread, and can be embedded in other data structures. Ports are the main message passing mechanism between threads in Oz.

Server-Clients Communication. The program shown in Figure 20 defines a thread that acts as FIFO queue server. Using logic variables makes the server insensitive to the arrival order of the `get` messages relative to `put` messages. A server is created by `{NewQueueServer Q}`. This statement returns back a port `Q` to the server. A client thread having access to `Q` can request services by sending message on the port. Notice how results are returned back through logic variables. A client requesting an Item in the queue will send the message `{Send Q get(I)}`. The server will eventually answer back by binding `I`.

The following sequence of statement illustrates the working of the program.

```
declare NewQueueServer in
local
   fun {NewQueue}
      X in q(front:X rear:X number:0)
   end
   fun {Put Q I}
      X in
      Q.rear = I|X
      {AdjoinList Q [rear#X number#(Q.number+1)]}
   end
   proc {Get Q ?I ?NQ}
      X in
      Q.front = I|X
      NQ = {AdjoinList Q [front#X number#(Q.number-1)]}
   end
   proc {Empty Q ?B ?NQ}
      B = (Q.number == 0) NQ = Q
   end
   proc {QServer Xs Q}
      case Xs
      of X|Xr then
         NQ in
         case X
         of put(I) then NQ = {Put Q I}
         [] get(I) then {Get Q I NQ}
         [] empty(B) then {Empty Q B NQ}
         else NQ = Q end
         {QServer Xr NQ}
      [] nil then skip
      end
   end
   S P = {NewPort S}
in fun {NewQueueServer}
      thread {QServer S {NewQueue}} end
      P
   end
end
```

Figure 20: Concurrent Queue Server

```
Q = {NewQueueServer}
{Send Q put(1)}
{Browse {Send Q get($)}}
{Browse {Send Q get($)}}
{Browse {Send Q get($)}}
{Send Q put(2)}
{Send Q put(3)}
{Browse {Send Q empty($)}}
```

Explain the use of logic variables as a mechanism to returns value.

Explain the common pattern of using nesting marking in message passing.

9.2 Chunks

Ports are actually stateful data structures. A port keeps a local state internally tracking the end of its associated stream.

Oz provides two primitive devices to construct abstract stateful data-types *chunks* and *cells*. All others subtypes of chunks can be defined in terms of chunks and cells.

A chunk is similar to a record except that the label of a chunk is an oz-name, and there is no arity operation available on chunks. This means one can hide certain components of a chunk if the feature of the component is a name that is visible only (by lexical scoping) to user-define operations on the chunk.

A chunk is created by the procedure `{NewChunk +Record}`. This create a chunk with the same arguments as the record, but having a unique label. The following program creates a chunk.

```
local X   in
   {Browse X={NewChunk f(c:3 a:1 b:2)}}
   {Browse X.c}
end
```

It will display the following.

```
<Ch>(a:1 b:2 c:3)
3
```

9.3 Cells and State

A cell could be seen as a chunk with a mutable single component. A cell is created as follows.

```
{NewCell X C}
```

A cell is created with an initial content referring to the variable `X`. The variable `C` contains the cell. The Table 2 shows the operations on a cell.

Check the following program. The last statement increments the cell by one. If we leave out **`thread ... end`** the program deadlocks. Do you know why?

```
local I O X
in
   I = {NewCell a} {Browse {Access I}}
   {Assign I b}    {Browse {Access I}}
   {Assign I X}    {Browse {Access I}}
   X = 5*5
   {Exchange I O thread O+1 end} {Browse {Access I}}
```

Procedure	Description
`{NewCell X C}`	creates a cell `C` with content `X`
`{Access +C X}`	returns the content of `C` in `X`
`{Assign +C Y}`	modifies the content of `C` to `Y`
`{IsCell +C}`	tests if `C` a cell
`{Exchange +C X Y}`	swaps the content of `C` from `X` to `Y`

Table 2: Cell Operations

end

Cells and higher-order iterators allow conventional assignment-based programming in Oz. The following program accumulates in the cell J the sum $1+2+3+\ldots+10$:

```
declare J in
J = {NewCell 0}
{For 1 10 1
   proc {$ I}
      O N in
      {Exchange J O N}
      N = O+I
   end }
{Browse {Access J}}
```

Explain how ports can be implemented by chunks and cells.

10 Classes and Objects

A Class in Oz is a chunk that contains:

- A collection of methods in a method table.
- A description of the attributes that each instance of the class will possess. An attribute is statefull cell that is accessed by the attribute-name which is a literal.
- A description of the features that each instance of the class will possess. A feature is an immutable component (a variable) that is accessed by the feature-name which is a literal.

Class are stateless oz-values[12]. Contrary to languages like SmallTalk, Java etc., they are just descriptions of how the objects of the class should behave.

Classes from First Principles. The Figure 21 example shows how a class is constructed from first principles. Here we construct a `CounterClass`. It has a single attribute accessed by the atom `val`. It has a method table that has three methods accessed through the record features `browse`, `init` and `inc`. A method is a procedure that takes a message, always a record, an extra parameter representing

```
declare Counter
local
   Attrs = [val]
   MethodTable = m(browse:MyBrowse init:Init inc:Inc)
   proc {Init M=init(Value) S Self}
      {Assign S.val Value}
   end
   proc {Inc M=inc(Value) S Self}
      local X in {Access S.val X} {Assign S.val X+Value} end
   end
   proc {MyBrowse M=browse S Self}
      {Browse {Access S.val}}
   end
in
   Counter = {NewChunk c(methods:MethodTable attrs:Atts)}
end
```

Figure 21: A Example of Class Construction

the state of the current object, and the object itself known internally as **`self`**. Talk about the example. Explain `M=init(...)`

Objects from First Principles. Figure 22 shows a generic procedure that create an object from a given class. This procedure creates an object state from the attributes of the class. It initializes the attributes of the object, each to a cell (with unbound initial value). We use here the iterator `Record.forAll` that iterates over all fields of a record. `NewObject` returns a procedure `Object` that identifies the object. Notice that the state of the object is visible only within `Object`. One may say that `Object` is a procedure that encapsulates the state[13].

We can try our program as follows

```
declare C
{NewObject Counter init(0) C}
{C inc(6)} {C inc(6)}
{C browse}
```

Try to execute the following statement.

```
local X in {C inc(X)} X=5 end {C browse}
```

You will see that nothing happens. The reason is the object call `{C inc(X)}` suspends inside the `inc` method. Do you know where exactly? If you on the other hand execute the following statement, things will work as expected.

```
local X in thread {C inc(X)} end X=5 end  {C browse}
```

[12] In fact classes may have some invisible state. In the current implementation a class usually has method cache which is stateful.

[13] This is a simplification; an object in Oz is a chunk that has the above procedure in one of its fields; other fields contain the object features.

```
declare
proc {NewObject Class InitialMethod ?Object}
   local
      State O
   in
      State = {MakeRecord s Class.attrs}
      {Record.forAll State proc {$ A} {NewCell _ A} end}
      proc {O M}
         {Class.methods.{Label M} M State O}
      end
      {O InitialMethod}
      Object = O
   end
end
```

Figure 22: Object Construction

10.1 Objects and Classes for Real

Oz supports object-oriented programming following the methodology outlined above. There is also syntactic support and optimized implementation so that object calls (calling a method in objects) is as cheap as procedure calls. The class `Counter` defined earlier has the syntactic form shown in Figure 23: A class X is defined by **`class`** X `...` **`end`**.
Attributes are defined using the attribute-declaration part before the method-declaration part:

`attr` A_1 `...` A_n

Then follows the method declarations, each has the form

`meth` E S **`end`**

where E evaluates to a method head which is a record whose label is the method name. An attribute A is accessed using the expression `@`A. It is assigned a value using the expression A `<-` `...`. The following shows how an object is created from a class using the procedure `New`, whose first argument is the class, the second is the initial method, and the result is the object. `New` is a generic procedure for creating objects from classes.

```
declare C in
C = {New Counter init(0)}
{C browse}
{C inc(1)}
local X in thread {C inc(X)} end X=5 end
```

10.2 Static Method Calls

Given a class $Class$ and a method head $m(\ldots)$, a method call has the following form:

$Class$ `,` $m(\ldots)$

```
declare
class Counter
   attr val
   meth browse
      {Browse @val}
   end
   meth inc(Value)
      val <- @val + Value
   end
   meth init(Value)
      val <- Value
   end
end
```

Figure 23: Counter Class

A method call invokes the method defined in the class argument. A method call can only be used inside method definitions. This is because a method call takes the current object **self** as implicit argument. The method could be defined there or inherited from super classes. Inheritance will be explained shortly.

Classes as Modules Static method calls have in general the same efficiency as procedure calls. This allows classes to be used as modules. This is advantageous because classes can be built incremently by inheritance. The program shown in Figure 24 shows a possible class acting as a module. The class `ListC` defines some common list-procedures as methods. `ListC` defines the methods `append/3`, `member/2`, `length/2`, and `nrev/2`. Notice that a method body is similar to any Oz statement, but in addition method calls are allowed. We also see the first instance of inheritance in

```
class ListC from BaseObject
```

Here the class ListC inherits from the predefined class `BaseObject`. `BaseObject` has only one trivial method:

```
meth noop skip end
```

To use a class as a module one need to create an object from it. This is done by:

```
declare ListM = {New ListC noop}
```

`ListM` is an object that will act as a module. We can try this module by performing some method calls:

```
{Browse {ListM append([1 2 3] [4 5] $)}}
{Browse {ListM length([1 2 3]  $)}}
{Browse {ListM nrev([1 2 3]  $)}}
```

10.3 Inheritance

Classes may inherit from one or several classes appearing after the keyword **from**. A class B is a *superclass* of a class A if B appears in the **from** declaration of A,

```
declare ListC in
class ListC from BaseObject
   meth append(Xs Ys ?Zs)
      case Xs
      of nil then Ys = Zs
      [] X|Xr then
         Zr in
         Zs = X|Zr
         ListC, append(Xr Ys Zr)
      end
   end
   meth member(X L ?B)
      {Member X L B}
   end
   meth length(Xs ?N)
      case Xs
      of nil then N = 0
      [] _|Xr then
         N1 in
         ListC, length(Xr N1)
         N = N1+1
      end
   end
   meth nrev(Xs ?Ys)
      case Xs
      of nil then Ys = nil
      [] X|Xr then
         Yr in
         ListC, nrev(Xr Yr)
         ListC, append(Yr [X] Ys)
      end
   end
end
```

Figure 24: List Class

or B is a superclass of a class appearing in the **from** declaration of A. Inheritance is a way to construct new classes from existing classes. It defines what attributes, features[14] and methods are available in the new class. We will restrict our discussion of inheritance to methods. However the same rules apply to features and attributes.

Which methods are available in a class is defined through a precedence relation on the methods that appear in a class hierarchy. We call this relation the overriding relation:

- A method in a class overrides any method (with the same label) in a super class.
- A method defined in a class declared in a **from** declaration overrrides any method (with the same label) defined in a class to the left in the **from** declaration.

Now a class hierarchy with the super-class relation can be seen as a directed graph with the class being defined as the root. There are two requirements for the inheritance to be valid. First, the inheritance relation is directed and acyclic. So the following is not allowed:

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

Second, after striking out all overriden methods each remaining method should have a unique label and is defined only in one class in the hierarchy. Hence class `C` in the following example is not valid because the two methods labeled `m` remain.

```
class A meth m(...) ... end end
class B meth m(...) ... end end
class B from B1 end
class A from A1 end
class C from A B end
```

Whereas in the following class `C` is valid and the method `m` available in `C` is that of `B`

```
class A meth m(...) ... end end
class B meth m(...) ... end end
class C from A B end
```

Notice that if you run a program with an invalid hierarchy, the system will not complain until an object is created that tries to access an invalid method. Only at this point of time you are going to get a runtime exception. The reason is that classes are partially formed at compile time. Rather they are completed by demand, using method caches, at execution time.

Multiple inheritance or not. My opinion is the following:

- In general, to really use multiple inheritance correctly, one has to understand the total inheritance hierarchy. Which is sometimes worth the effort.
- I do not like the existance of any overriding rule that depends on the order of classes in the **from** construct. So I would consider the later example as invalid

[14] to be defined shortly

as the former one. The reason is that if you take `A` and `B` in the later example, and refine them to get the first example your working program will suddenly cease to work.

- If there is a name conflict between immediate super classes I would define a method locally that overrides the conflict-causing methods.
- There is another problem with multiple inheritance when sibling super-classes share (directly or indirectly) a common ancestor-class that is stateful (i.e. has attributes). One may get replicated operations on the same attribute. This typically happens when initializing a class, one has to initialize its super classes. The only remedy here is to carefully understand the inheritance hierarchy to avoid such replication. Or, inherit from multiple classes that do not share common ancestor. This problem is known as the implementation-sharing problem.

10.4 Features

Objects may have features similar to records. Features are components that are specified in the class declaration:

```
class C from ...
   feat a1 ... an
end
```

As in a record a feature of an object has an associated field. The field is a logic variable that can be bound to any Oz value (including cells, objects, classes etc.). Features of objects are accessed using the infix ´.´ operator. The following shows an example using features:

```
declare
class ApartmentC from BaseObject
    meth init skip end
end
class AptC from ApartmentC
   feat
      streetName: york
      streetNumber:100
      wallColor:white
      floorSurface:wood
end
```

Feature initialization. The example shows how features could be initialized at the time the class is defined. In this case all instances of the class `AptC` will have the features of the class, with the same values of the features. Therefore

```
declare Apt1 Apt2
Apt1 = {New AptC init}
Apt1 = {New AptC init}
{Browse Apt1.streetName}
{Browse Apt2.streetName}
```

will display `york` twice. We may leave a feature uninitialized as in:

```
declare
class MyAptC1 from ApartmentC
   feat
      streetName
end
```

In this case whenever an instance is created, the field of the feature is assigned a new fresh variable. Therefore

```
declare Apt3 Apt4
Apt3 = {New MyAptC1 init}
Apt4 = {New MyAptC1 init}
Apt3.streetName = kungsgatan
Apt4.streetName = sturegatan
```

will assign the feature `streetName` of object `Apt3` the value `kungsgatan`, and the corresponding feature of `Apt4` the value `sturegatan`.

One more form of initialization is available. A feature is initialized at the class declaration to a variable or an Oz-value that has a variable. In the following:

```
declare
class MyAptC1 from ApartmentC
   feat
      streetName:f(_)
end
```

the attribute is initialized to a tuple with an anonymous variable. In this case all instances of the class will *share* the same variable. So the following:

```
declare Apt1 Apt2
Apt1 = {New MyAptC1 init}
Apt2 = {New MyAptC1 init}
{Browse Apt1.streetName}
{Browse Apt2.streetName}
Apt1.streetName = f(york)
```

if entered incrementely, will show that the statement `Apt1.streetName = f(york)` binds the corresponding feature of `Apt2` to the same value as that of `Apt1`.

10.5 Parameterized Classes

There are many ways to get your classes more generic that can later be specialized for specific purposes. The common way in object-oriented programming is to define first *an abstract class* where some methods are left unspecified. Later these methods are defined in subclasses of this class. Suppose you defined a generic class for sorting where the comparison operator `lessThan` is needed. This operator depends on what kind of data are being sorted. Different realizations are needed for integer, rational, or complex numbers. In this case, by subclassing we can specialize the abstract class to a class for a 'concrete' type.

In Oz we have also another natural method for creating generic classes. Since classes are first-class values, we can instead define a function the takes some type argument(s) and return a class that is specialized for the type(s). In Figure 25 a function `SortClass` is defined that takes a class as its single argument and returns a sorting class specialized for the argument.

```
declare
fun {SortClass Type}
   class C from BaseObject
      meth qsort(Xs Ys)
         case Xs
         of nil then Ys = nil
         [] P|Xr then
            S L in
            {self partition(Xr P S L)}
            ListC, append(C,qsort(S $)  P|(C,qsort(L $)) Ys)
         end
      end
      meth partition(Xs P Ss Ls)
         case Xs
         of nil then Ss = nil Ls = nil
         [] X|Xr then
            Sr Lr in
            case Type,less(X P $) then
               Ss = X|Sr Lr = Ls
            else
               Ss = Sr Ls = X|Lr
            end
            C,partition(Xr P Sr Lr)
         end
      end
   end
in
   C
end
```

Figure 25: Parameterized Classes

We can now define two classes for integers and rationals:

```
declare
class Int
   meth less(X Y B)
      B = X<Y
   end
end
class Rat from Object
   meth less(X Y B)
      local
         ´/´(P Q) = X
         ´/´(R S) = Y
      in
         B = P*S < Q*R
      end
   end
end
```

Thereafter, we can execute the following statements:

```
{Browse {{New {SortClass Int} noop} qsort([1 2 5 3 4] $)}}
{Browse {{New {SortClass Rat} noop}
         qsort([´/´(23 3) ´/´(34 11) ´/´(47 17)] $)}}
```

Self Application. The program in Figure 25 shows in the method `qsort` an object application using the keyword **`self`**:

```
meth qsort(Xs Ys)
   case Xs
     ...
      {self partition(Xr P S L)}
     ...
```

We use here the phrase object-application instead of the commonly known phrase message-sending because message sending is misleading in a concurrent language like Oz. When we use **`self`** instead of a specific object as in `{self partition(Xr P S L)}` we mean that we dynamically pick the method `partition` that is defined in the current object. Thereafter we apply the object (as a procedure) on the message. This is a form of dynamic binding common in all object-oriented languages.

10.6 Attributes

We have touched before on the notion of attributes. Attributes are the carriers of state in objects. Attributes are declared similar to features, but by using the keyword **`attr`**. When an object is created each attributes is assigned a new cell as its value. These cells are initialized very much the same way as features. The difference lies in the fact that attributes are cells that can be assigned, reassigned an accessed at will. However, attributes are private to their objects. The only way to manipulate an attribute from outside an object, is to force the class designer to write a method that manipulates the attribute.

In the Figure 26 we define an the class `Point`. Note that the attributes `x` and `y` are initialized to `0` before the initial message is applied. The method `move` uses

```
declare
class Point from BaseObject
   attr x:0 y:0
   meth init(X Y)
      x <- X
      y <- Y     % attribute update
   end
   meth location(L)
      L = l(x:@x y:@y) %attribute access
   end
   meth moveHorizontal(X)
      x <- X
   end
   meth moveVertical(Y)
      y <- Y
   end
   meth move(X Y)
      {self moveHorizontal(X)}
      {self moveVertical(Y)}
   end
   meth display
      {Browse {StringToAtom{
               VirtualString.toString "point at ("#@x#" ,
"#@y#")"}}}
   end
end
```

Figure 26: The class Point

self application internally.

Try to create an instance of `Point` and apply some few messages:

```
declare P
P = {New Point init(2 0)}
{P display}
{P move(3 2)}
```

10.7 Private and Protected Methods

Methods may be labelled by variables instead of literals. These methods are *private* to the class in whicg they are defined, as in:

```
class C from ...
  meth A(X) ... end
  meth a(...) {self A(5)} ... end
  ....
end
```

The method `A` is visible only within the class `C`. In fact the notation above is just an abbreviation of the following expanded definition:

```
local A = {NewName} in
```

```
   class C from ...
      meth A(X) ... end
      meth a(...) {self A(5)} ... end
      ...
   end
end
```

where A is bound to a new name in the lexical scope of the class definition.

Some object-oriented languages have also the notion of protected methods. A method is *protected* if it is accessible only in the class it is defined or in descendant classes, i.e. subclasses and subsubclasses etc. In Oz there is no direct way to define a method to be protected. However there is a programming technique that gives the same effect. We know that attributes are only visible inside a class or to descendants of a class by inheritance. We may make a method protected by firstly making it private and secondly storing it is in an attribute. Consider the following example:

```
class C from ...
  attr pa:A
  meth A(X) ... end
  meth a(...) {self A(5)} ... end
  ....
end
```

Now we subclass `C` and access method `A` as follows:

```
class C1 from C
  meth b(...) {self @pa(5)} ... end
  ....
end
```

Method `b` access method `A` through the attribute `pa`.

Let us continue our simple example in Figure 26 by defining a specialization of the class that in addition of being a point, it stores a histroy of the previous movement. This is shown in Figure 27.

There are a number of remarks on the class definition `HistoryPoint`. First observe the typical pattern of method refinement. The method `move` specializes that of class `Point`. It first calls the super method, and then does what is specific to being a history-point class. Second method `DisplayHistory` is made private to the class. Moreover it is made available for subclasses, i.e. protected, by storing it in the the attribute `displayHistory`. You can now try the class by the following statements:

```
declare P
P = {New HistoryPoint init(2 0)}
{P display}
{P move(3 2)}
```

10.8 Default Argument Values

A method head may have default argument values. So in the following example:

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

a method call or object application of `a` may leave the arguments of features `d1` and `d2` unspecified. In this case these arguments will assume the value `0`.

```
declare
class HistoryPoint from Point
   attr
      history: nil
      displayHistory: DisplayHistory
   meth init(X Y)
      Point,init(X Y)  % call your super
      history <- [l(X Y)]
   end
   meth move(X Y)
      Point,move(X Y)
      history <- l(X Y)|@history
   end
   meth display
      Point,display
      {self DisplayHistory}
   end
   meth DisplayHistory  % private method
      {Browse ´with location history: ´}
      {Browse @history}
   end
end
```

Figure 27: The class HistoryPoint

We can go back to our `Point` example by specialized `Point` in a different direction. We define the class `BoundedPoint` as a point that moves in a constrained rectangular area. Any attempt to move such a point outside of the area will be ignored. The class is shown in Figure 28. Notice that the method `init` has two default arguments that gives a default area if not specified in the initialization of an instance of `BoundedPoint`.

We conclude this section by finishing our example in a way that shows the multiple inheritance problem. We would like now a specialization of both `HistoryPoint` and `BoundedPoint` as a bounded-history point. A point that keeps track of the history and moves in a constrained area. We do this by defining the class `BHPoint` that inherits from the two previously defined classes. Since they both share the class `Point` which contains stateful attributes we encounter the implementation-sharing problem. Since we in any way anticipated this problem, we created two protected methods stored in `boundConstraint` and `displayHistory` to avoid repeating the same actions. In any case we had to refine the methods `init`, `move` and `display` since they occur in the two sibling classes. The solution is shown in Figure 29. Notice how we use the protected methods. We did not care avoiding the repetition of initializing the attributes `x` and `y` since it does not make so much harm. Try the following example:

```
declare P
P = {New BHPoint init(2 0)}
{P display}
```

```
declare
class BoundedPoint from Point
   attr
      xbounds: 0#0
      ybounds: 0#0
      boundConstraint: BoundConstraint
   meth init(X Y xbounds:XB <= 0#10 ybounds:YB <= 0#10)
      Point,init(X Y) % call your super
      xbounds <- XB
      ybounds <- YB
   end
   meth move(X Y)
      case {self BoundConstraint(X Y $)} then
         Point,move(X Y)
      else skip end
   end
   meth BoundConstraint(X Y B)
      B = (X >= @xbounds.1 andthen
           X =< @xbounds.2 andthen
           Y >= @ybounds.1 andthen
           Y =< @ybounds.2 )
   end
   meth display
      Point,display
      {self DisplayBounds}
   end
   meth DisplayBounds
      X0#X1 = @xbounds
      Y0#Y1 = @ybounds
      S
   in
      S = "xbounds=("#X0#","#X1#"), ybounds=("#Y0#","#Y1#")"
      {Browse {StringToAtom {VirtualString.toString S}}}
   end
end
```

Figure 28: The class BoundedPoint

```
declare
class BHPoint from HistoryPoint BoundedPoint
   meth init(X Y xbounds:XB <= 0#10 ybounds:YB <= 0#10)
      HistoryPoint,init(X Y)
      BoundedPoint,init(X Y xbounds:XB ybounds:YB) % repeats init
   end
   meth move(X Y)
      L in L = @boundConstraint
      case {self L(X Y $)} then
         HistoryPoint,move(X Y)
      else skip end
   end
   meth display
      BoundedPoint,display
      {self @displayHistory}
   end
end
```

Figure 29: The class BHPoint

```
{P move(1 2)}
```

This pretty much covers the object system. What is left is how to deal with concurrent threads sharing common space of objects.

11 Concurrent Objects

As we have seen objects in Oz are stateful data structures. Threads are the active computation entities. Threads can communicate either by message passing using ports, or through common shared objects. Communication through shared objects requires the ability to serialize concurrent operations on objects so that the object state is kept coherent after each such operation. In Oz we separate the issue acquiring exclusive access of an object from the object system. This gives us the ability to perform coarse-grain atomic operation of a set of objects. A very important requirement in distributed database system. The basic mechanism in Oz to getting exclusive access is through locks.

11.1 Locks

The purpose of a lock is to mediate exclusive access to a shared resource between threads. Such a mechanism is typically made safer and more robust by restricting this exclusive access to a critical region: on entry into the region, the lock is secured and the thread is granted exclusive access rights to the resource, and when execution leaves the region, whether normally or through an exception, the lock is released. Typically, a concurrent attempt to secure the same lock will block until the thread currently holding it has released it.

Simple Locks. In the case of a simple lock, a nested attempt by the same thread to secure the same lock during the dynamic scope of a critical section guarded by it will block: reentrancy is not supported. Simple locks can be modelled in Oz as follows, where `Code` is a nullary procedure encapsulating the computation to be performed in the critical section.

```
proc {NewSimpleLock ?Lock}
   Cell = {NewCell unit}
in
   proc {Lock Code}
      Old New
   in {Exchange Cell Old New}
      {Wait Old}
      try {Code} finally New=unit end
   end
end
```

Thread-Reentrant Locks. In Oz, the computational unit is the thread. Therefore an appropriate locking mechanism should grant exclusive access rights to threads. As a consequence the non-reentrant simple lock mechanism presented above is inadequate. A thread-reentrant lock allows the same thread to reenter the lock, i.e. to enter a dynamically nested critical region guarded by the same lock. Such a lock can be secured by at most one thread at a time. Concurrent threads that attempt to secure the same lock are queued. When the lock is released, it is granted to the thread standing first in line etc. Thread-reentrant locks can be modelled in Oz as follows:

```
proc {NewLock ?Lock}
   Token   = {NewCell unit}
   Current = {NewCell unit}
in proc {Lock Code}
      ThisThread = {Thread.this}
   in case
         ThisThread == {Access Current}
      then
         {Code}
      else
         Old New
      in
         {Exchange Token Old New}
         {Wait Old}
         {Assign Current ThisThread}
         try {Code}
         finally
            {Assign Current unit} New=Old
         end
      end
   end
end
```

Locks in Oz are Thread-reentrant locks that are given syntactic and implementational support in Oz. They are implemented as subtype chunks. Oz provides the following syntax for guarded critical regions:

```
declare A L in
A = {NewArray 1 10 ~5}
L = {NewLock}
proc {Switch A}
   {For {Array.low A} {Array.high A} 1
    proc {$ I}
       X in
       X = {Get A I}
       case X<0 then {Put A I ~X}
       elsecase X == 0  then {Put A I zero} else skip end
       {Delay 100}
    end}
end
proc {Zero A}
   {For {Array.low A} {Array.high A} 1
    proc {$ I} {Put A I 0} {Delay 100} end}
end
```

Figure 30: Locking

`lock` *E* **`then`** *S* **`end`**

where *E* is an expression and the construct blocks until *E* is determined. If E is not a lock, then a type error is raised. `{NewLock L}` allocates a new lock `L`. `{IsLock E}` returns true iff `E` is a lock.

Arrays. Oz has arrays as primary chunk type. Operation on arrays are defined in module `Array` [2]. To create an array the procedure `{NewArray +L +H I A} is used where` \A? is the resulting array, `L` is the lower-bound index, `H` is the higher-bound index, and `I` is the initial value. As a simple illustration of locks consider the program in Figure 30.

Assume that we want to perform the procedures `Zero` and `Switch` each atomically but in an arbitrary order. To do this we can use locks as in the following example:

```
local X Y in
thread {Delay 100} lock L then {Zero A} end  X = unit end
thread lock L then {Switch A} end Y = X end
{Wait Y} {For 1 10 1 proc {$ I} {Browse {Get A I}} end}
end
```

By Switching the delay statement above between the the first and the second threads, we observe that all the elements of the array either will get the value `zero` or `0`. We have no mixed values.

Write an example of an atomic transaction on multiple objects.

11.2 Locking Objects

To guarantee mutual exclusion on objects one may use the locks described in the pervious subsection. Alternatively we may declare in the class of the objects that its instances can be locked with a default lock existing in the object when it is created. A class with a lock is declared as follows:

```
class C from ....
  prop locking
  ....
end
```

This does not automatically lock the object when one of its methods are called instead we have to use the construct:

```
lock S end
```

inside a method to guarantee exclusive access when `S` is executed.

Remember that our locks are thread-reentrant. This implies that

- if we take all objects that we constructed and enclose each method body with `lock ... end`, and
- execute our program with only one thread, then
- the program will behave exactly as before

Of course if we use multiple threads we might deadlock if there is any cyclic dependency. Writing nontrivial concurrent program needs careful understanding of the dependancy patterns between threads. In such programs deadlock may occur whether locks are used or not. It suffices to have a cyclic communication pattern for deadlock to occur.

The program in Figure 23 can be refined to work in concurrent environment by refining it as follows:

```
declare
class CCounter from Counter
   prop locking
   meth inc(Value)
      lock Counter,inc(Value) end
   end
   meth init(Value)
      lock Counter,init(Value) end
   end
end
```

Let us now study a number of interesting examples where threads not only perform atomic transactions on objects, but also synchronize through objects. The first example shows a concurrent channel which is shared among an arbitrary number of threads. A producing thread may put information in the channel asynchronously. A consuming thread has to wait until information exists in the channel. Waiting threads are served fairly. Figure 31 shows one possible realization. This program relies on the use of logical variables to achieve the desired synchronization. The method `put` inserts an element in the channel. A thread executing the method `get` will wait until an element is put in the channel. Multiple consuming threads will

```
declare
class Channel from BaseObject
   prop locking
   attr f r
   meth init
      X in f <- X r <- X
   end
   meth put(I)
      X in lock @r=I|X r<-X end
   end
   meth get(?I)
      X in lock @f=I|X f<-X end {Wait I}
   end
end
```

Figure 31: An Asynchronous Channel Class

reserve their place in the channel, thereby achieving fairness. Notice that `{Wait I}` is done outside an exclusive region. If waiting was done inside `lock ... end` the program would deadlock. So, as a rule of thumb do not wait inside an exclusive region, if the waking-up action has to acquire the same lock.

The next example shows a traditional way to write *monitors*. In Oz we do not use the normal signaling mechanism of monitors currently known as $notify(event)$ and $waitForEvent(event)$. Instead logic variables gives the same effect. We show here an example of a unit buffer. The unit buffer behaves in a very similar way as channel when it comes to conumers. In the case of producers only one is allowed to insert an item in the an empty buffer. Other producers have to suspend until the item is consumed. The program is Figure 32 shows a single buffer monitor. Here we had to program a signalling mechanism for producers in a way similar to traditional monitors. Observe the pattern in the `put` method. First waiting has to be done outside the exclusive region. This is done by using an auxiliary variable `X` which gets bound to `wait` if waiting has to be done. The signaling mechanism is using a channel stored in the attribute `prodq`. The `get` method notifies one producer at a time by seting the `empty` flag and notifying one producer (if any). This is done as an atomic step. Notice that this example uses reentrant locking in an essential way. Do you how?

The example discussed above was an illustration of how to write traditional monitors. There is a more simple way to write a unit buffer class that is not traditional. This is due to the combination of objects and logic varaible Here follows in Figure UnitBuffer2 a simple definition of the `UnitBuffer` class no locking is needed directly.

Simple generalization of this leads to an arbitrary size bounded buffer class. This is shown in Figure 34.

```
declare
class UnitBuffer from Channel
   prop locking
   attr empty:true prodq
   meth init
      Channel, init
      pq <- {New Channel init}
   end
   meth put(I)
      X in
      lock
         case @empty then
            Channel,put(I) empty<-false
            X = nowait
         else X=wait end
      end
      case X==wait then
         {@prodq get(_)} {self put(I)}
      else skip end
   end
   meth get(I)
      Channel, get(I)
      lock empty <- true {@prodq put(unit)} end
   end
end
```

Figure 32: A Unit Buffer Monitor

```
declare
class UnitBuffer from BaseObject
   attr putq getq
   meth init
      putq <- {New Queue init}
      getq <- {New Queue init}
      {@getq put(unit)}
   end
   meth put(I)
      {@getq get(_)}
      {@putq put(I)}
   end
   meth get(?I)
      {@putq get(I)}
      {@getq put(unit)}
   end
end
```

Figure 33: A Unit Buffer Class

```
\begin{ozdisplay}
declare
class BoundedBuffer from BaseObject
   attr putq getq
   meth init(N)
      putq <- {New Queue init}
      getq <- {New Queue init}
      {For 1 N 1 proc {$ _} {@getq put(unit)} end}
   end
   meth put(I)
      {@getq get(_)}
      {@putq put(I)}
   end
   meth get(?I)
      {@putq get(I)}
      {@getq put(unit)}
   end
end
```

Figure 34: A Bounded Buffer Class

12 Distributed Programming

See the paper on mobile objects

13 Encapsulated Search

To be completed

14 Logic Programming

To be completed

15 Constraint Programming

See the Oz-primer

References

[1] V. Saraswat, *Concurrent Constraint Programming.*

[2] *The Oz Standard Modules*

[3] *The Oz Notation*

[4] *The Oz User Manual*