Tutorial of Oz

Seif Haridi
Nils Franzén

mozart

Abstract

This tutorial introduces the Oz programming language and the Mozart programming system. Oz is a multi-paradigm language that is designed for advanced, concurrent, networked, soft real-time, and reactive applications. Oz provides the salient features of object-oriented programming including state, abstract data types, objects, classes, and inheritance. It provides the salient features of functional programming including compositional syntax, first-class procedures/functions, and lexical scoping. It provides the salient features of logic programming and constraint programming including logic variables, constraints, disjunction constructs, and programmable search mechanisms. It allows users to dynamically create any number of sequential threads. The threads are dataflow threads in the sense that a thread executing an operation will suspend until all operands needed have a well-defined value.

The tutorial covers most of the concepts of Oz in an informal way. It is suitable as first reading for programmers who want to be able to quickly start writing programs without any particular theoretical background. The document is deliberately informal and thus complements other Oz documentation.

The Mozart programming system has been developed by researchers from DFKI (the German Research Center for Artificial Intelligence), SICS (the Swedish Institute of Computer Science), the University of the Saarland, UCL (the Université catholique de Louvain), and others.

The material in this document is still incomplete and subject to change from day to day.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

Introduction

The Mozart system implements Oz 3, the latest in the Oz family of multi-paradigm languages based on the concurrent constraint model. Oz 3 is almost completely upward compatible with its predecessor Oz 2. The main additions to Oz 2 are functors (a kind of software component) and futures (for improved dataflow behavior). Oz 2 is itself a successor to the original Oz 1 language, whose implementation was first released publicly in 1995. Except as otherwise noted, all references to Oz in the Mozart documentation are to Oz 3.

Oz 3 and the Mozart system have been developed mainly by the research groups of Gert Smolka at the DFKI (the German Research Center for Artificial Intelligence), Seif Haridi at SICS (the Swedish Institute of Computer Science), and Peter Van Roy at UCL (the Université catholique de Louvain).

Underlying all versions of Oz is a concurrent constraint programming model, extended to support stateful computations, i.e., computations on mutable objects. The theoretical foundation of the concurrent constraint model is given in [3]. The original Oz computation model, Oz 1, supports a fine-grained notion of concurrency where each statement can potentially be executed concurrently. This results in a fine-grained model similar to the actor model. A good exposition of the Oz 1 programming model is given in [5]. Our experience using Oz 1 showed that this kind of model, while theoretically appealing, makes it very hard for the programmer to control the resources of his/her application. It is also very hard to debug programs and the object model becomes unnecessarily awkward.

Oz 2 remedies these problems by using instead a thread-based concurrency model, with explicit creation of threads. A powerful new object system has been designed and traditional exception handling constructs have been added. In addition, the constraint solving and search capabilities have been greatly enhanced.

Oz 3 conservatively extends Oz 2 with two concepts, *functors* and *futures*, and also corrects several minor syntactic problems. A functor is a kind of software component. It specifies a module in terms of the other modules it needs. This supports incremental construction of programs from components that may be addressable over the Internet by URLs, see [1]. A future is a logic variable that can be read but not written. This allows safe dataflow synchronization over the Internet.

The Mozart system supports distributed and networked applications. It is possible to connect Oz computations located on different sites, resulting in a single network-transparent computation. Mozart supports automatic transfer of stateless data and code among sites, mobile computation (objects), message passing, shared logic variables

and orthogonal mechanisms for fault detection and handling for the network and for sites.

1.1 Summary of Oz features

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 programmers and system developers with a wide range of programming abstractions to enable them to develop complex applications quickly and robustly. Oz merges several directions of programming language designs into a single coherent design. Most of us know the benefits of the various programming paradigms whether object-oriented, functional, or constraint logic programming. When we start writing programs in any existing language, we quickly find ourselves confined by the concepts of the underlying paradigm. Oz solves this problem by a coherent design 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. More concretely:

- Oz combines the salient features of object-oriented programming, by providing state, abstract data types, classes, objects, and inheritance.

- Oz provides the salient features of functional programming by providing a compositional syntax, first-class procedures, and lexical scoping. In fact, every Oz entity is first class, including procedures, threads, classes, methods, and objects.

- Oz provides the salient features of logic programming and constraint programming by providing logic variables, disjunctive constructs, and programmable search strategies.

- Oz is a concurrent language where users can create dynamically any number of sequential threads that can interact with each other. However, in contrast to conventional concurrent languages, each Oz thread is a dataflow thread. Executing a statement in Oz proceeds only when all *real* dataflow dependencies on the variables involved are resolved.

- The Mozart system supports network-transparent distribution of Oz computations. Multiple Oz sites can connect together and automatically behave like a single Oz computation, sharing variables, objects, classes, and procedures. Sites disconnect automatically when references between entities on different sites cease to exist.

- In a distributed environment Oz provides language security. That is, all language entities are created and passed explicitly. An application cannot forge references nor access references that have not been explicitly given to it. The underlying representation of the language entities is inaccessible to the programmer. This is a consequence of having an abstract store and lexical scoping. Along with first-class procedures, these concepts are essential to implement a capability-based security policy, which is important in open distributed computing.

1.2 The Kernel Language

This section gives a short but precise introduction to the Oz kernel language. The full Oz language can be regarded as syntactic sugar to a small kernel language. The kernel language represents the essential part of the language.

Figure 1.1 The Oz kernel language

$$\langle \text{Statement} \rangle \quad ::= \quad \langle \text{Statement1} \rangle \; \langle \text{Statement2} \rangle$$

```
⟨Statement⟩   ::=   ⟨Statement1⟩ ⟨Statement2⟩
             |     X = f(l1:Y1 ... ln:Yn)
             |     X = ⟨number⟩
             |     X = ⟨atom⟩
             |     X = ⟨boolean⟩
             |     {NewName X}
             |     X = Y
             |     local X1 ... Xn in S1 end
             |     proc {X Y1 ... Yn} S1 end
             |     {X Y1 ... Yn}
             |     {NewCell Y X}
             |     Y=@X
             |     X:=Y
             |     {Exchange X Y Z}
             |     if B then S1 else S2 end
             |     thread S1 end
             |     try S1 catch X then S2 end
             |     raise X end
```

The Oz execution model consists of dataflow threads observing a shared store. Threads contain statement sequences Si and communicate through shared references in the store. A thread is *dataflow* if it only executes its next statement when all the values the statement needs are available. If the statement needs a value that is not yet available, then the thread automatically blocks until it can access that value. As we shall see, data availability in the Oz model is implemented using logic variables. The shared store is not physical memory, rather it is an abstract store which only allows operations that are legal for the entities involved, i.e., there is no direct way to inspect the internal representations of entities. The store contains unbound and bound logic variables, cells (named mutable pointers, i.e., explicit state), and procedures (named lexically scoped closures that are first-class entities). Variables can reference the names of procedures and cells. Cells point to variables. The external reference procedures are variables. When a variable is bound, it disappears, that is, all threads that reference it will automatically reference the binding instead. Variables can be bound to any entity, including other variables. The variable and procedure stores are monotonic, i.e., information can only be added to them, not changed or removed.

Figure 1.1 defines the abstract syntax of a statement S in the kernel language. We briefly define each possible statement. Statement sequences are reduced sequentially inside a thread. Values (records, numbers, etc.) are introduced explicitly and can be equated to variables. All variables are logic variables, declared in an explicit scope defined by the **local** statement. Procedures are defined at run-time with the **proc** statement and referred to by a variable. Procedure applications block until their first

argument refers to a procedure. State is created explicitly by *NewCell*, which creates a cell, an updateable pointer into the variable store. Cells are read by `@` and updated by `:=` or alternatively by *Exchange*. Conditionals use the keyword `if` and block until the condition variable *B* is `true` or `false` in the variable store. Threads are created explicitly with the `thread` statement. Exception handling is dynamically scoped and uses the `try` and `raise` statements.

The full Oz language is defined by transforming all its statements into this kernel language. This will be explained in detail in this document. Oz supports idioms such as objects, classes, reentrant locks, and ports [5][7]. The system implements them efficiently while respecting their definitions. As an introduction we will give a brief summary of each idiom's definition. For clarity, at this stage we have made small conceptual simplifications. Full definitions are given later in this document.

1.3 Classes

A class is essentially a record that contains the method table and attribute names. A class is defined through multiple inheritance, and any conflicts are resolved at definition time when building its method table.

1.4 Objects

An object is essentially a special record having a number of components. One component is the object's class. Another component is a one-argument procedure that references a cell, which is hidden by lexical scoping. The cell holds the object's state. Applying an object `Obj` to message `M` applies the object's procedure to `M`. The argument indexes into the method table. A method is a procedure that is given a reference to the state cell. In general it modifies the state of the object.

1.5 Reentrant locks

A reentrant lock is essentially a one-argument procedure `{Lck P}` used for explicit mutual exclusion, e.g., of method bodies in objects used concurrently. Reentrant locks use cells and logic variables to achieve their behavior. `P` is a zero-argument procedure defining the critical section. Reentrant means that the same thread is allowed to reenter the lock. Calls to the lock may therefore be nested. The lock is released automatically if the thread in the body terminates or raises an exception that escapes the lock body.

1.6 Ports

A port is an asynchronous channel that supports many-to-one communication. A port `P` encapsulates a stream `S`. A stream is a list with unbound tail. The operation `{Send P M}` adds `M` to the end of `S`. Successive sends from the same thread appear in the order they were sent.

The Interactive Development Environment

This tutorial contains many code examples and you are highly encouraged to try them out interactively as you go. This can be done very comfortably by taking advantage of the Mozart system's interactive development environment. We normally call it the OPI, which stands for the *Oz Programming Interface*, and it is described extensively in *"The Oz Programming Interface"*. In the present section, you will learn just enough about the OPI to allow you to start experimenting with our code examples.

2.1 Starting The OPI

Under Unix, the OPI is normally started by invoking the command `oz` at the shell prompt. Under Windows, the installation procedure will have provided you with a Mozart system program group: click on the Mozart item in this group. Shortly thereafter you get a window that looks like this:
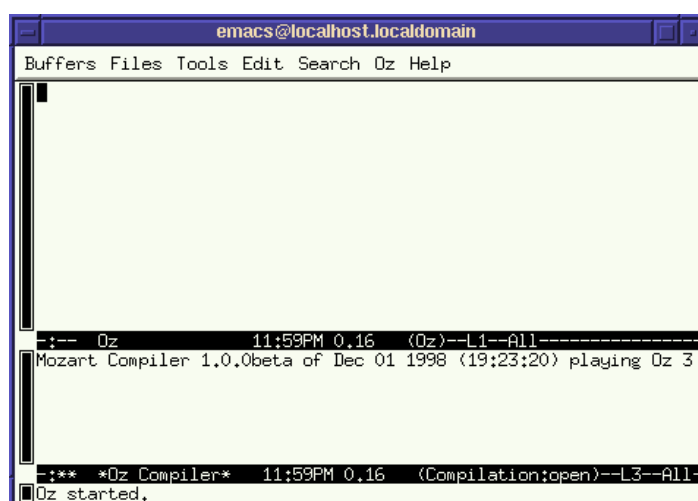
The OPI uses the Emacs editor as the programming front-end. If you are not familiar with Emacs or its terminology, you should consult the Emacs on-line tutorial [6] available from the Help menu in the Emacs menu bar.

The initial window is split in two text buffers. The upper buffer called `Oz` is a space where you can write small pieces of code and interactively execute them: it essentially

plays, for Oz code, the same role as the `*scratch*` buffer for emacs lisp code. The lower text buffer is called `*Oz Compiler*` and shows a transcript of your interaction with the compiler of the Mozart subprocess.

2.2 Hello World

Let us begin with the traditional *Hello World* example. In the `Oz` buffer, type the following:

```
{Show 'Hello World'}
```

This example illustrates the unconventional syntax of procedure invocation in Oz: it is indicated by curly braces. Here, procedure `Show` is invoked with, as single argument, the atom `'Hello World'`.

In order to execute this fragment, we position the point on the line we just typed and select *Feed Line* from the Oz menu in the menubar. We now see:
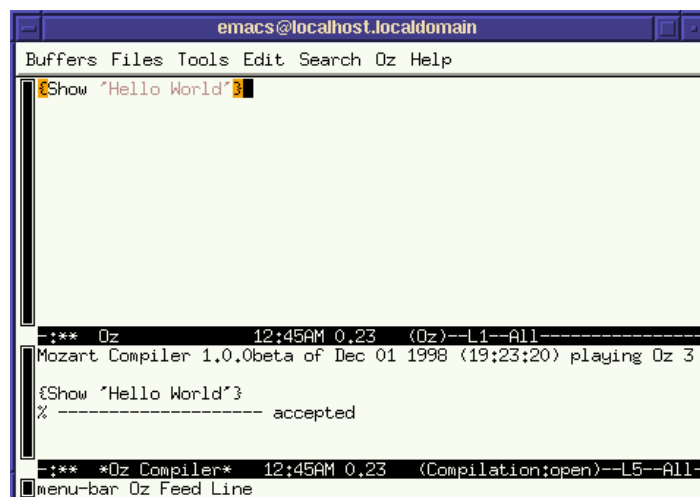
The transcript from the compiler indicates that `{Show 'Hello World'}` was *fed* to the compiler and *accepted*, i.e. successfully parsed and compiled. But was it executed, and, if yes, where is the output? Indeed it was executed, but the output appears in a different buffer called `*Oz Emulator*`: this contains the execution transcript. If we select from the Oz menu *Show/Hide -> Emulator*, we now see:
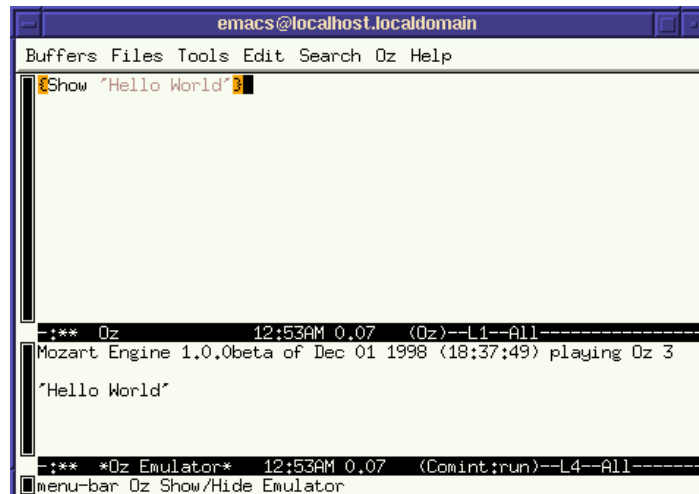
2.3 Good News For The Programmer

The OPI has many features to support interactive code development.

2.3.1 Code Editing

The `oz-mode` is a major mode for editing Oz code, and provides automatic indentation as well `font-lock` support for code colorization.

2.3.2 Key Bindings

You may interact with the underlying Mozart subprocess from any buffer in `oz-mode`, not just from the `Oz` buffer as demonstrated earlier. Furthermore, all the actions that we carried out in the *Hello World* example can be invoked more conveniently through key bindings instead of through the Oz menu.

`C-. C-l`	Feed current line
`C-. C-r`	Feed selected region
`C-. C-b`	Feed whole buffer
`M-C-x`	Feed current paragraph
`C-. C-p`	*idem*
`C-. c`	Toggle display of `*Oz Compiler*` buffer
`C-. e`	Toggle display of `*Oz Emulator*` buffer

a 'paragraph' is a region of text delimited by empty lines.

2.3.3 Compiler Errors

The OPI also has support for conveniently dealing with errors reported by the compiler. Let us type the following erroneous code in the `Oz` buffer:

```
local A B in
   A = 3
   proc {B}
      {Show A + 'Tinman'}
   end
   {B 7}
end
```

and feed it to the compiler using M-C-x. The compiler reports 2 errors and we see:

C-x ` (that is *Control-x backquote*) positions the transcript to make the first error message visible and moves the point, in the source buffer, to where the bug is likely to be located.

Indeed, we should not try to add an integer and an atom! If we invoke C-x ` once more, we see:
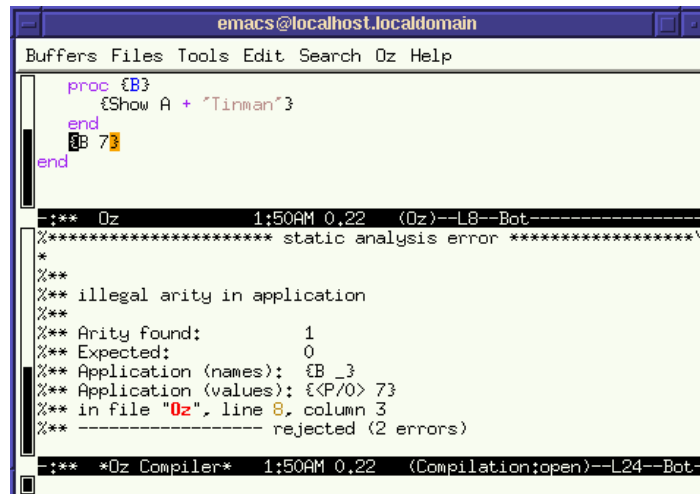
Here, we have mistakenly applied a nullary procedure to an argument.

2.3.4 Graphical Development Tools

The mozart system has many graphical tools. Here we only mention the *Browser* which is otherwise extensively documented in *"The Oz Browser"*. So far, we merely used the procedure `Show` to print out values. Instead, we can invoke `Browse` to get a graphical display interface. For example, feeding:

```
{Browse 'Hello World'}
```

causes the following new window to pop up:
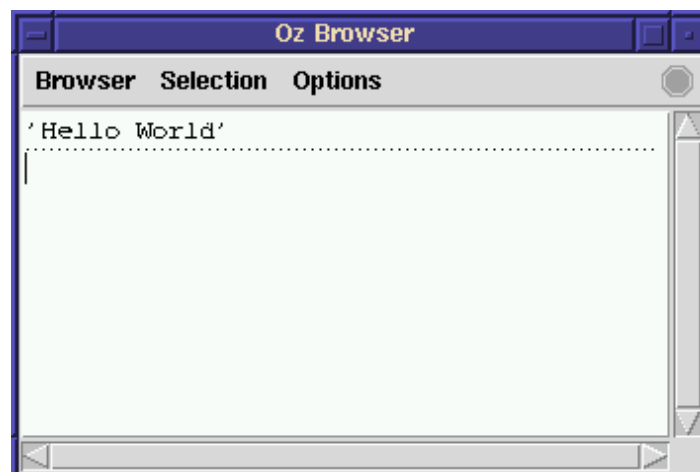
This is not very exciting, but let's now feed this code:

```
declare W H
{Browse foo(width:W height:H surface:thread W*H end)}
```

Now the browser window shows a term that is only partially known (instantiated) since variable `W` and `H` have been declared but not yet bound to values:

```
┌─────────────────────────────────────────────┐
│ ─  │              Oz Browser          │ □ │ ▫ │
├─────────────────────────────────────────────┤
│ Browser  Selection  Options            ◯    │
├─────────────────────────────────────────────┤
│ 'Hello World'                          △    │
│...........................................  │
│ foo(                                        │
│     height:H                                │
│     surface:_                               │
│     width:W)                                │
│...........................................  │
│ |                                           │
│                                             │
│                                             │
│                                        ▽    │
│ ◁                                      ▷    │
└─────────────────────────────────────────────┘
```

Now let us feed `W=3` and we see that the browser automatically updates the display to reflect the information we just added.

```
┌─────────────────────────────────────────────┐
│ ─  │              Oz Browser          │ □ │ ▫ │
├─────────────────────────────────────────────┤
│ Browser  Selection  Options            ◯    │
├─────────────────────────────────────────────┤
│ 'Hello World'                          △    │
│...........................................  │
│ foo(height:H surface:_ width:3)             │
│...........................................  │
│ |                                           │
│                                             │
│                                             │
│                                             │
│                                        ▽    │
│ ◁                                      ▷    │
└─────────────────────────────────────────────┘
```
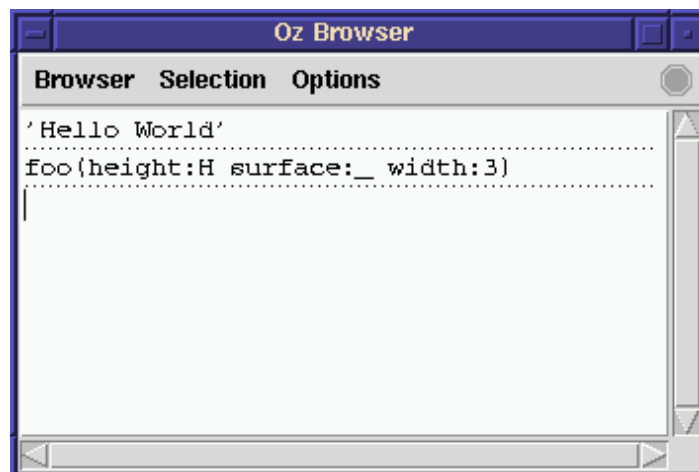
Now we feed `H=5` and again the browser updates the display and now shows a fully instantiated term:

```
┌─────────────────────────────────────────────┐
│ ─  │              Oz Browser          │ □ │ ▫ │
├─────────────────────────────────────────────┤
│ Browser  Selection  Options            ◯    │
├─────────────────────────────────────────────┤
│ 'Hello World'                          △    │
│...........................................  │
│ foo(height:5 surface:15 width:3)            │
│...........................................  │
│ |                                           │
│                                             │
│                                             │
│                                             │
│                                        ▽    │
│ ◁                                      ▷    │
└─────────────────────────────────────────────┘
```
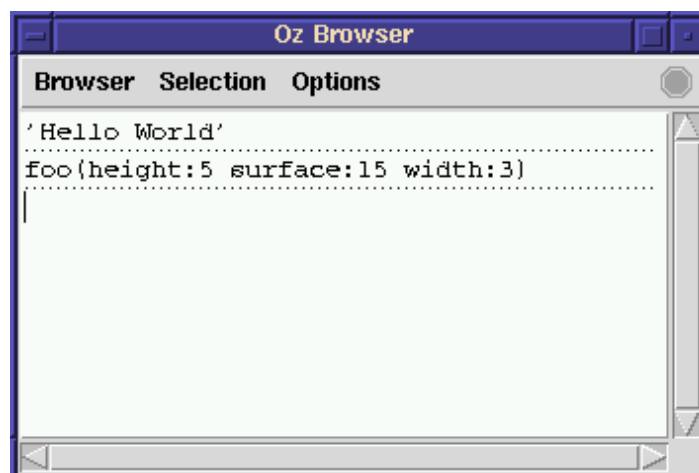
The browser allows you to see the evolution of the instantiation of a term as concurrent computations (threads) proceed and add more information.

Basics

We will initially restrict ourselves to the sequential programming style of Oz. At this stage you may think of Oz computations as performed by a sequential process that executes one statement after the other. We call this process a *thread*. This thread has access to the *store*. It is able to manipulate the store by reading, adding, and updating information stored in the store. Information is accessed through the notion of *variables*. A thread can access information only through the variables visible to it, directly or indirectly. Oz variables are *single-assignment* variables or more appropriately logic variables. In imperative languages like C and Java, a variable can be assigned multiple times. In contrast, single assignment variables can be assigned only once. This notion is known from many languages including dataflow languages and concurrent logic programming languages. A single assignment variable has a number of phases in its lifetime. Initially it is introduced with unknown value, 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. A *logic variable* is a single assignment variable that can also be equated with another variable. Using logic variables does not mean that you cannot model state-change because a variable, as you will see later, could be bound to a cell, which is stateful, i.e., the content of a cell can be changed.

A thread executing the statement:

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

will introduce three single assignment variables `X`, `Y`, `Z` and execute the statement `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. Variables may also be presented textually as any string of printable characters enclosed within back-quote characters, e.g. `` `this $ is a variable` ``. Before the execution of `S` the variables declared will not have any associated values. We say that 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`, as well as in all statements that follow `S` textually, unless overridden again by another variable declaration of the same textual variables. `X`, `Y`, `Z` are global variables.

3.1 Primary Oz Types

Figure 3.1 Oz Type Hierarchy
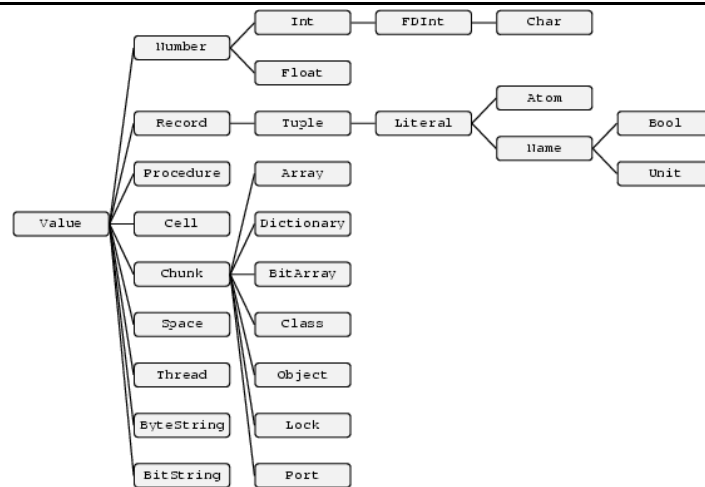
Oz is a dynamically typed language. Figure 3.1 shows the type hierarchy of Oz. Any variable, if it ever gets a value, will be bound to a value of one of these types. Most of the types seem familiar to experienced programmers, except probably *Chunk*, *Cell*, *Space*, *FDInt* and *Name*. We will discuss all of these types in due course. For the impatient reader here are some hints. The *Chunk* data type allows users to introduce new abstract data types. *Cell* introduces the primitive notion of state-container and state modification. *Space* will be needed for advanced problem solving using search techniques. *FDInt* is the type of finite domain that is used frequently in constraint programming, and constraint satisfaction. *Name* introduces anonymous unique unforgeable tokens.

The language is dynamically-typed in the sense that when a variable is introduced, its type as well as its value are unknown. Only when the variable is bound to an Oz value, does its type become determined.

3.2 Adding Information

In Oz, there are few ways of adding information to the store or (said differently) of binding a variable to a value. The most common form is using the *equality* infix operator =. For example, given that the variable x is declared the following statement:

```
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. to any other value different from 1, a proper *exception* will be raised. Exception handling is described later.

3.3 Data Types with Structural Equality

The hierarchy starting from *Number* and *Record* in Figure 3.1 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. For example, if both are integers and are identical numbers or both are lists and their head elements are identical as well as their respective tail lists. Structural equality allows values to be equivalent even if they are replicas occupying different physical memory location.

3.4 Numbers

The following program, introduces three variables `I`,`F` and `C`. It assigns `I` an integer, `F` a float, and `C` the character `t` in this order. It then 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 supports 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 decimal points. Other examples of floats are shown where `~` is unary minus:

```
~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 [4]. Characters are a subtype of integers in the range of `0, ..., 255`. The standard ISO 8859-1 coding is used (not Unicode). Printable characters have external representation, e.g. `&0` is actually the integer `48`, and `&a` is `97`. Some control characters have also a representation e.g. `&\n` is a new line. All characters can be written as `&\`*ooo*, where *o* is an octal digit.

Operations on characters, integers, and floats can be found in the library modules `Char`[1], `Float`[2], and `Int`[3]. Additional generic operations on all numbers are found in the module `Number`[4].

---

[1] Section *Characters*, *(The Oz Base Environment)*
[2] Section *Floats*, *(The Oz Base Environment)*
[3] Section *Integers*, *(The Oz Base Environment)*
[4] Section *Numbers in General*, *(The Oz Base Environment)*

3.5   Literals

Another important category of atomic types, i.e. types whose members have no internal structure, is the category of literals. Literals are divided into atoms and names. An Atom is symbolic entity that has an identity made up of a sequence of alphanumeric characters starting with a lower case letter, or arbitrary printable characters enclosed in quotes. For example:

```
a   foo   '='   ':='   'OZ 3.0'   'Hello World'
```

Atoms have an ordering based on lexicographic ordering.

Another category of elementary entities is `Name`. 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 worldwide 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` that consists of the single name **unit**. This is used as synchronization token in many concurrent programs.

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

3.6   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 are `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 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)
```

It is just a syntactic notation for the record:

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

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)]
```

3.7 Operations on records

We discuss some basic operations on records. Most operations are found in the module Record[5]. To select a field of a record component, we use the infix dot operator, e.g. *Record.Feature*

```
% Selecting a Component
{Browse T.key}
{Browse W.1}
% will show seif twice in the browser
seif
seif
```

The *arity* of a record is a list of the features of the record sorted lexicographically. To display the arity of a record we use the procedure Arity. The procedure application {Arity R X} will execute once R is bound to a record, and will bind X to the arity of the record. Executing the following statements

```
% 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 display

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

---

[5]Section *Records in General, (The Oz Base Environment)*

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, X is bound to R.F, otherwise X is bound to the default value D. 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 the pair 1#(2#3). With the # operator, you cannot directly write an empty or a single element tuple. Instead, you must fall back on the usual prefix record syntax: the empty tuple must be written '#'() or just '#', and a single element tuple '#'(X).

The operation {AdjoinAt R1 F X R2} binds R2 to the record resulting from adjoining the field X to R1 at feature F. If R1 already has the feature F, the resulting record R2 is identical to R1 except for the field R1.F whose value becomes X. Otherwise the argument F:X is added to R1 resulting in R2.

The operation {AdjoinList R LP S} takes a record R, a list of feature-field pairs, and returns in S a new record such that:

- The label of R is equal to the label of S.

- S has the components that are specified in LP in addition to all components in R that do not have a feature occurring in LP.

This operation is of course defined by using AdjointAt.

```
local S in
   {AdjoinList tree(a:1 b:2) [a#3 c#4] S}
   {Show S}
end
% gives S=tree(a:3 b:2 c:4)
```

3.8 Lists

As in many other symbolic programming languages, e.g. Scheme and Prolog, *lists* form an important class of data structures in Oz. The category of lists does not belong to a single data type in Oz. They are rather a conceptual structure. 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 positive integers is represented as:

    1|2|3|nil

Another convenient special notation for a *closed list*, i.e. a list with a 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 `1` and `2`, but whose tail is the variable `x` looks like:

    1|2|X

One can also use the standard record 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 3.0"

is the list

    [79 90 32 51 46 48]

or equivalently

    [&O &Z & &3 &. &0]

3.9 Virtual Strings

A virtual string is a special tuple that represents a string 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 `'#'`, as well as numbers, strings, or `'#'`-labeled tuples can be used to compose virtual strings. Here is one example:

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

represents the string

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

For each data type discussed in section, there is a corresponding module in the Mozart system. The modules define operations on the corresponding data type. You may find more about these operations in The Oz Base Environment documentation[6]

---

[6]*"The Oz Base Environment"*

Equality and the Equality Test Operator

We have so far shown simple examples of the equality statement, e.g.

```
W = tree(I Y LT LR)
```

These were simple enough to understand intuitively 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 memory words called *nodes*. Each node is labeled by a logic variable. When a variable `X` is introduced a new node is created in the store, labeled by `X`, having the value *unknown*. At this point, the node does not possess any real value; it is empty as a container that may be filled later.

A variable labeling a node whose value is *unknown* is an *unbound* variable. The nodes are 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 node associated with `W`. Notice that we are just getting a graph structure. The node contains a record with four fields. The fields contain arcs pointing to the nodes labeled by `I`, `Y`, `LT`, and `LR` respectively. Each arc, in turn, is labeled by the corresponding feature of the record. Given two variables `X` and `Y`, the operation `X = Y` will try to *merge* their respective nodes. Now we are in a position to give a reasonable account for the merge operation `X = Y`, known as the *incremental tell* or alternatively the *unification* operation.

- If X and Y label the same node, the operation is completed successfully.

- If X (resp. Y) is unbound then merge the node of X (resp. Y) with the node of Y (resp. X). Merging means replacing all references to the node X by a reference to Y[1]. Conceptually the original node of X has been discarded.

- If X and Y label different nodes containing the records Rx and Ry respectively:

---

[1]This could be done by many various ways. One way is to let the node X point to the node Y, and changing X to be a reference node. The chain of reference node are always traversed before performing any unification operation.

– If *Rx* and *Ry* have different labels, arities, or both: the operation is completed, and an exception is raised.

– Otherwise, the arguments of *Rx* and *Ry* with the same feature are pair-wise merged in arbitrary order.

In general the two graphs, to be merged, could have cycles. However any correct implementation of the merge operation will remember the node pairs for which an attempt to merge has been made earlier, and considers the operation to be successfully performed. A more formal description of the incremental tell operation is found in [2].

When a variable is no longer accessible, a process known as garbage collection reclaims its node.

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.

To be able to see the finite representation of Z, you have to switch the Browser to *Minimal Graph* presentation mode. Choose the Option menu, Representation field, and click on Minimal Graph.

The Browser is described in *"The Oz Browser"*.

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

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

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

4.1   Equality test operator `==`

The basic procedure `{Value.'==' X Y R}` tries to test whether `X` and `Y` are equal or not, and returns the result in `R`.

• It returns the Boolean value **true** if the graphs starting from the nodes of `X` and `Y` have the same structure, with each pair-wise corresponding nodes having identical Oz values or are the same node.

- It returns the Boolean value **false** if the graphs have different structure, or some pair-wise corresponding nodes have different values.

- It suspends when it arrives at pair-wise corresponding nodes that are different, but at least one of them is unbound.

Now remember this, if a procedure suspends, the whole thread suspends! This does not seem very useful. However, as you will see later, it becomes a very useful operation when multiple threads start interacting with each other.

The equality test is normally used as a functional expression, rather than a statement. {Value.'==' X Y R} can also be written R = X**==**Y using the infix **==** operator. This is further illustrated in the example below:

```
% 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
```

Basic Control Structures

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

>   *S1*   *S2*

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 *S1*, before starting *S2*. In fact, *S2* may not be executed at all, if an exception is raised in *S1*.

5.1 skip

The statement **skip** is the empty statement.

5.2 If Statement

Oz provides a simple form of conditional statement having the following form:

>   **if** B **then** *S1* **else** *S2* **end**

B should be a Boolean value.

5.2.1 Semantics

- If B is bound to **true** *S1* is executed
- if B is bound to **false** *S2* is executed
- if B is bound to an non-boolean value, an exception is raised
- otherwise if B is unbound the thread suspends until one of the cases above applies

---

Figure 5.1 Using a if statement

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

Comparison Procedures Oz provides a number of built-in tertiary procedures used for comparison. These include `==` that we have seen earlier as well as `\=`, `=<`, `<`, `>=`, and `>`. Common to these procedures is that they are used as Boolean functions in an infix notation. The following example illustrates the use of an If-statement in conjunction with the greater-than operator `>`.

In this example `Z` is bound to the maximum of `X` and `Y`, i.e. to `Y`:

5.2.2 Abbreviations

A statement using the keyword `elseif`:

```
if B1 then S1 elseif B2 then S2 else S3 end
```

is shorthand for nested if-statements:

```
if B1 then S1
else if B2 then S2
     else S3 end
end
```

An if-statement missing the else part:

```
if B1 then S1 end
```

is equivalent to:

```
if B1 then S1 else skip end
```

5.3 Procedural Abstraction

5.3.1 Procedure Definition

Procedure definition 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.

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

5.3.2 Semantics

Assume that the variable P is already introduced; executing the above statement will:

- create a unique closure which is essentially a uniquely named lambda expression $\lambda(X_1 \ldots X_N).S$

- The variable P is bound to the closure.

A procedure in Oz has a unique identity, given by its unique closure, 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 equality. Others include threads, cells, and chunks.

5.4   On Lexical Scoping

In general, the statement S in a procedure definition will have many variable occurrences. A variable that occurs textually in a statement is called an identifier to distinguish it from the logic variable that is a data structure created at runtime. Some identifier occurrences in S are *syntactically bound* while others are *free*. An identifier occurrence X^1 in S is bound if it is in the scope of the procedure formal-parameter X, or is in the scope of a variable introduction statement that introduces X. Otherwise, the identifier occurrence is free. Each free identifier occurrence in a program is eventually bound by the closest textually surrounding identifier-binding construct.

We have already seen how to apply (call) a procedure. Let us now show our first procedure definition. In Figure 5.1, 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}
      if X >= Y then Z = X else Z = Y end
   end
   X = 5
   Y = 10
   {Max X Y Z} {Browse Z}
end
```

5.5   Anonymous Procedures and Variable Initialization

One could ask why a variable is bound to a procedure in a way that is different from it being bound to a record, e.g. X = *f(...)*? The answer is that what you see is just a syntactic variant of the equivalent form

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

---

[1]This rule is approximate, since class methods and patterns bind identifier occurrences

The R.H.S. defines an *anonymous procedural value*. This is equivalent to

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

In Oz, we can initialize a variable immediately while it is being introduced by using a *variable-initialization equality*

```
X = ⟨Value⟩
```

or

```
⟨Record⟩ = ⟨Value⟩
```

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}
              if 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 the ones being introduced. Consider the following example:

```
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 variables on the L.H.S. are introduced, i.e. M, Y, X1, and L. Therefore the outer variable 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 intro-duction of the inner `Y` in the L.H.S. of the initializing equality by using an exclamation mark `!` as follows. An exclamation mark `!` is only meaningful in the L.H.S. of an initializing equality [2].

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

5.6   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 5.2. The symbol `?` before `TreeOut` is a voluntary *documentation comment* denoting that the argument plays the role of an output argu-ment. 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 in the tree and the input key. Notice the use of **if ... then ... el-seif ... else ... end** with the obvious meaning.

In Figure 5.2, the local variable introduction statement

```
local tree(K1 V1 T1 T2)= TreeIn in ...
```

---

[2]In fact the exclamation mark `!` can be used in other situation where you want to suppress the intro-duction of new variables, for example in pattern matching constructs

Figure 5.2 Inserting a node (key and value) in a binary tree

```
proc {Insert Key Value TreeIn ?TreeOut}
   if TreeIn == nil then TreeOut = tree(Key Value nil nil)
   else
      local tree(K1 V1 T1 T2) = TreeIn in
         if Key == K1 then TreeOut = tree(Key Value T1 T2)
         elseif 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
end
```

performed implicitly a pattern matching to extract the values of the locally introduced variables `K1`, `V1`, `T1` and `T2`.

Oz provides an explicit pattern-matching case statement, which allows implicit introduction of variables in the patterns.

5.6.1 Case Statement

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

All variables introduced in *Pattern_i* are implicitly declared, and have a scope stretching over the corresponding *Si*.

5.6.2 Semantics

Let us assume that expression `E` is evaluated to `V`. Executing the case statement will sequentially try to match `V` against the patterns *Pattern_1*, *Pattern_2*, ...,*Pattern_n* in this order. Matching `V` against `Pattern_i` is done in left-to-right depth-first manner.

- If `V` matches `Pattern_i` without binding any variable occuring in `V`, the corresponding *Si* statement is executed.

- If `V` matches `Pattern_i` but binds some variables occuring in `V`, the thread suspends

- If the matching of `V` and `Pattern_i` fails, `V` is tried against the next pattern `Pattern_i+1`, otherwise the **else** statement *S* is executed.

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 a new local variable by using **!**. For example, in the following example:

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

`X1` is matched is against the value of the external variable `Y`. Now remember again that the case statement and its executing thread may suspend if `X1` is insufficiently instantiated to decide the result of the matching. Having all this said, Figure 5.3 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}
```

Figure 5.3 Tree insertion using case statement

```
% case for pattern matching
proc {Insert Key Value TreeIn ?TreeOut}
   case TreeIn
   of nil then TreeOut = tree(Key Value nil nil)
   [] tree(K1 V1 T1 T2) then
      if Key == K1 then TreeOut = tree(Key Value T1 T2)
      elseif 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
```

The expression *E* we may match against, could be any record structure, and not just a variable. This allows multiple argument matching, as shown in Figure 5.4, which expects two sorted lists `Xs` and `Ys` and merges them into a sorted list `Zs`.

Figure 5.4 Merging of two sorted lists

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

5.7 Nesting

Let us use our `Insert` procedure as defined in Figure 5.3. 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 ... $ ...} ... }
```

Using $ as a *nesting marker*, and thereby 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.

5.7.1   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 `R`. Therefore `{P X ...}` could be 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|{SMerge Xr Ys}
```

Here, the nested application goes *after* the record (or list) construction statement. Therefore, we get

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

Doing so makes many recursive procedures be *tail-recursive*. Tail-recursive procedures execute with the space efficiency of iterative constructs.

We can now rewrite our `SMerge` procedure as shown in Figure 5.5, where we use nested application.

5.8   Procedures as Values

Since we have been inserting elements in binary trees, let us define a program that checks if a data structure is actually a binary tree. The procedure `BinaryTree` shown in Figure 5.6 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}`. 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 a new procedure

Figure 5.5 Merging two sorted lists written in nested form

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

Figure 5.6 Checking a binary tree

```
% What is a binary tree?
local
   proc {And B1 B2 ?B}
      if B1 then B = B2 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
```

AndThen that takes as its first two arguments two procedures, and calls the second procedure only if the first returns **false**; thus, getting the effect of delaying the evaluation of its arguments until really needed. The procedure is shown Figure 5.7. AndThen is the first example of *a higher-order procedure*, i.e. a procedure that takes other procedures as arguments, and may return other procedures as results. In our case, AndThen just returns a Boolean value. However, 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 help creating generic reusable components.

Figure 5.7 Checking a binary tree lazily

```
local
   proc {AndThen BP1 BP2 ?B}
      if {BP1} then B = {BP2} 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
```

5.9   Control Abstractions

Higher-order procedures are used in Oz to define various control abstractions. In the modules `Control`[3] and `List`[4] as well as many others, you will find many control abstractions. Here are some examples. The procedure {For From To Step P}[5] 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`. Executing {For 1 10 1 Browse} will display the integers 1 2 **...** 10.

Figure 5.8 The For iterator

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

Another control abstraction that is often used is the `ForAll/2` iterator defined in the `List` module. `ForAll/2` applies a unary procedure on all the elements of a list, in

---

[3]Chapter *Control, (The Oz Base Environment)*
[4]Section *Lists, (The Oz Base Environment)*
[5]Section *Loops, (The Oz Base Environment)*

the order defined by the list. Think what happens if the list is produced incrementally by another concurrent thread? In this case the consumer thread will synchronize on the availability of data on the list. The list behaves as a stream of elements and we automatically get stream communication between threads.

```
proc {ForAll Xs P}
   case Xs
   of nil then skip
   [] X|Xr then
      {P X}
      {ForAll Xr P}
   end
end
```

5.10 Exception Handling

Oz incorporates an exception handling mechanism that allows safeguarding programs against exceptional and/or unforeseeable situations at run-time. It is also possible to raise and handle user-defined exceptions.

An exception is any expression *E*. To raise the exception *E*, one executes the following statement:

```
raise E end
```

Here is a simple example:

```
proc {Eval E}
  case E
  of   plus(X Y) then {Browse X+Y}
  []   times(X Y) then {Browse X*Y}
  else raise illFormedExpression(E) end
  end
end
```

The basic exception handling statement is called a try-statement. Its simplest form is:

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

Execution of this statement is equivalent to executing *S1* if *S1* does not raise an exception. If *S1* raises an exception *E*, *X* gets bound to *E* and the statement *S2* is executed. The variable *X* is visible in the scope of *S2*.

A more convenient try statement has the following form:

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

This is equivalent to:

```
try S catch X then
   case X
   of Pattern_1 then S1
   [] Pattern_2 then S2
      ...
   [] Pattern_n then Sn
   else raise X end end
end
```

Put into words, the Execution of this statement is equivalent to executing *S* if *S* does not raise an exception. If *S* raises exception *E* and *E* matches one of the patterns *Pattern_i*, control is passed to the corresponding statement *S_i*. If *E* does not match any pattern the exception is propagated outside the try-statement until eventually caught by the system, which catches all escaping exceptions.

```
try
    {ForAll [plus(5 10) times(6 11) min(7 10)] Eval}
catch
    illFormedExpression(X) then {Browse '** '#X#' **'}
end
```

A try-statement may also specify a final statement *S_final*, which is executed on normal as well as on exceptional exit.

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

Assume that `F`[6] is an opened file; the procedure `Process/1` manipulates the file in some way; and the procedure `CloseFile/1` closes the file. The following program ensures that the `F` is closed upon normal or exceptional exit.

```
try
   {Process F}
catch X then {Browse '** '#X#' **'}
finally {CloseFile F} end
```

---

[6]We will now see how input/output is handled later

5.11   System Exceptions

The exceptions raised by the Oz system are records with one of the labels: `failure`, `error`, and `system`.

- `failure`: indicates the attempt to perform an inconsistent equality operation on the store of Oz.

- `error`: indicates a runtime error which should not occur such as applying a nonprocedure to some argument or adding an integer to an atom, etc.

- `system`: indicates a runtime condition because of the environment of the Mozart operating system process, i.e., an unforeseeable situation like a closed file or window; or failing to open a connection between two Mozart processes.

The exact format of Mozart system-exceptions is in an experimental state and therefore the user is advised to rely only on the label, as in the following example:

```
proc {One X} X=1 end
proc {Two X} X=2 end
try {One}={Two}
catch
   failure(...) then {Show caughtFailure}
end
```

Here the pattern `failure(...)` catches any record whose label is `failure`. When an exception is raised but not handled, an error message is printed in the emulator window (standard error), and the current thread terminates. In stand-alone applications the default behavior is that a message is printed on standard error and the whole application terminates. It is possible to change this behavior to something else that is more desirable for particular applications.

6

Functions

6.1 Functional Notation

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

```
{P X1 ... Xn R}
```

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

```
{P X1 ... Xn}
```

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

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

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

```
proc {F X1 ... Xn R} S R=E end
```

The exact syntax for functions as well as their transformation into procedure definitions is defined in the *The Oz Notation Reference Manual*[1].

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 6.1 is the functional equivalent to the program shown in Figure 5.7. Notice how the function `AndThen/2` is unfolded into the procedure `AndThen/3`. Below we show a number of steps that give some intuition of the transformation process. All the intermediate forms are legal Oz programs.

---

[1] *"The Oz Notation"*

```
fun {AndThen BP1 BP2}
   if {BP1} then {BP2}
   else false end
end
```

Make a procedure by introducing a result variable B:

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

Move the result variable into the outer *if-expression* to make it an *if-statement*:

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

Figure 6.1 Checking a binary tree lazily

```
% Syntax Convenience: functional notation
local
   fun {AndThen BP1 BP2}
      if {BP1} then {BP2}
      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
```

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 functional languages [2] including Standard ML, Scheme, and the concurrent functional language Erlang. However, standard function definitions in Oz are not lazy. Lazy functions are also supported in Oz[3].

Here is an example of the well-known higher order function `Map/2`. It is tail recursive in Oz but not in Standard ML or in Scheme.

---

[2]Strict functional languages evaluate all its argument first before executing the function

[3]We will discuss them later when talking about *futures* and by need synchronization

```
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}}
```

6.1.1 `andthen` and `orelse`

After all, we have been doing a lot of work for nothing! Oz already provides the Boolean lazy (non-strict) versions of the functions `And/2` and `Or/2` as the Boolean operators **andthen** and **orelse** respectively. The former behaves like the function `AndThen/2`, 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 6.2 defines the final version of the function `BinaryTree`.

Figure 6.2 Checking a binary tree lazily

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

6.1.2 To Function or not to function?

Since now, in principal, we have some syntactic redundancy by either using procedures or functions, 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 `SMerge` example. Moreover, 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 non-functional definition due to stateful data types or nondeterministic definitions[4].

- 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. After all functions are concise.

---

[4]In fact, in those cases the use of the object-oriented style of Oz is most appropriate

Modules and Interfaces

Modules, also known as packages, are collection of procedures and other values[1] 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 provide the external services of the module. In Oz there is syntactic support for module specification. The concept used is called *functor* . A functor is an expression that specifies the components of a module. The Mozart system converts a functor to a module with the help of a module manager.

Let us first see what a module is, and then look to a corresponding functor that specifies the module. In general a module is a bunch of locally defined entities, e.g. procedures, objects, accessible through a record interface. 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 {MergeSort ...} ... end
   proc {Sort ... } ... {MergeSort ...} ... end
   proc {Member ...} ... end
 in
   List = 'export'(append: Append
                   sort: Sort
                   member: Member
                   ... )
end
```

Access to Append procedure outside of the module List is done by using the field append from the *record* List: List.append. Notice that in the above example the procedure MergeSort is private to the module. Most of the base library modules of Mozart follow the above structure. The above module can be created from a functor that looks as follows:

```
functor
export
   append:Append
```

---

[1]Classes, objects, etc.

```
      sort:Sort
      member:Member
      ...
   define
      proc {Append ... }  ... end
      proc {MergeSort ...} ... end
      proc {Sort ... } ... {MergeSort ...} ... end
      proc {Member ...} ... end
   end
```

Assuming that this functor is stored, somehow, on the file '/home/xxx/list.ozf', the module can be created as follows:

```
   declare [List]= {Module.link ['/home/xxx/list.ozf']}
```

Module.link/2 is a function defined in the module Module that takes a list of functors, links them together, returns a corresponding list of modules.

Functors may also have import declarations. If you want to import a system module you can just state the name of its functor. On the other-hand importing a user-defined module requires stating the URL of the file where the functor is stored.

Consider the following functor.

```
   functor
   import
      Browser
      FO at 'file:///home/seif/FileOperations.ozf'
   define
      {Browser.browse {FO.countLines '/etc/passwd'}}
   end
```

The **import** declaration imports the system module Browser, and uses the procedure Browser.browse. It also imports the module FO specified by the functor stored in the file '/home/seif/FileOperations.ozf', and calls the procedure FO.countLines which counts the number of lines in a file given as argument. This functor is defined for its effect, therefore it does not export any interface. When this functor is linked the statement between **define ... end** is executed.

Given a file 'x.oz' defining a functor, you may create the corresponding functor 'x.ozf' from your shell by typing the command:

```
   ozc -c x.ozf
```

Concurrency

So far, we have seen only one thread executing. It is time to introduce concurrency. In Oz a 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 non-terminating thread, that is not blocking, will eventually be allocated a time slice of the processor. This means that threads are executed fairly.

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

Each thread has a unique name. To get the name of the current thread the procedure `Thread.this/1` is called. Having a reference to a thread, by using its name, enables operations on threads such as terminating a thread, or raising an exception in a thread. Thread operations are defined the base module `Thread`.

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 in
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. Observe now what happens when you 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.

Figure 8.1 A concurrent Map function

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

The program shown in Figure 8.1 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
F X Y Z
{Browse thread {Map X F} end}
```

a thread executing `Map` is created. It will suspend immediately in the case-statement because `X` is unbound. If we thereafter 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 it will suspend 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 8.2 is a concurrent divide-and-conquer program, which is rather inefficient way to compute the *Fibonacci* function. This program creates an exponential number of threads! See how easy it is to create concurrent threads. You may use this program to test how many threads your Oz installation can create. Try

```
{Browse {Fib 25}}
```

Figure 8.2 A concurrent Fibonacci function

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

Figure 8.3 The Mozart Panel showing thread creation {Fib 26 X}

---
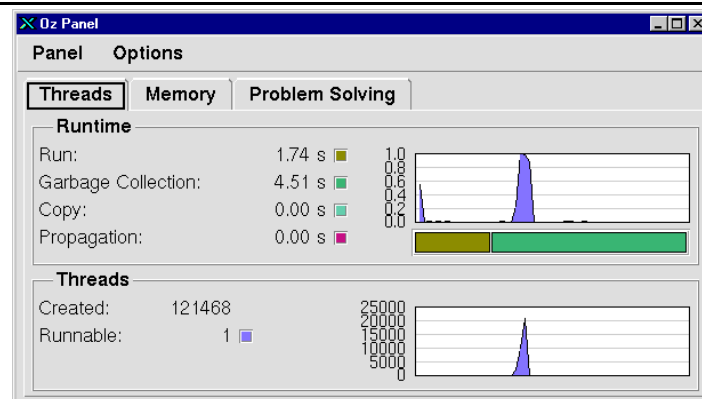
while using the panel program in your Oz menu to see the threads. If it works, try a larger number. The panel is show in Figure 8.3.

The whole idea of explicit thread creation in Oz is to enable the programmer to structure his/her application in a modular way. In this respect the Mozart system is excellent. Threads are so cheap that can afford to create say 100000 of them. As a comparison thread creation in Mozart 1.0 is about 60 time faster than in Java JDK 1.2. If concurrency makes an easier structure of you program then use it without hesitation. However sequential programs are always faster than concurrent programs having the same structure. The `Fib` program in Figure 8.2 is faster if you remove `thread ... end`. Therefore, create threads only when the application needs it, and not because concurrency is fun.

8.1   Time

In module `Time`[1], we can find a number of useful soft 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`.

---

[1] Section *Time*, *(The Oz Base Environment)*

Figure 8.4 A 'Ping Pong' program

```
local
   proc {Ping N}
      if 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
```

The program shown in Figure 8.4 starts two threads, one displays `ping` periodically after 500 milliseconds, and the other `pong` after 600 milliseconds. Some `pings` will be displayed immediately after each other because of the periodicity difference.

8.1.1 Making Standalone Application

It is easy to make stand-alone applications in Mozart. We show this by make the program in Figure 8.4 stand-alone by making a functor of the program as shown in Figure 8.5, and storing it in your file `PingPong.oz`. Thereafter use the command:

```
ozc -x PingPong.oz
```

Now type PingPong in your shell to start the program.[2]

8.2 Stream Communication

The data-flow property of Oz easily enables writing threads that communicate through streams in a producer-consumer pattern. A stream is a list that is created incrementally by one thread (the producer) and subsequently consumed by one or more threads (the consumers). The threads consume the same elements of the stream. For example, the program in Figure 8.6 is an example of stream communication, where the producer generates a list of numbers and the consumer sums all the numbers.

Try the program above by running the following program:

```
{Browse thread {Sum thread {Generator 150000} end } end}
```

---

[2]To terminate this program in the OS shell you have to type CONTROL-C. We will see later how to terminate it properly.

Figure 8.5 A 'Ping Pong' program stand-alone

```
functor
import
   Browser(browse:Browse) %Import Browse form Browser module
define
   proc {Ping N}
      if 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 8.6 Summing the elements in a list

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

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

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

The consumer waits on the stream until items arrive, then the items are 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* suspends the consumer 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
```

Figure 8.7 shows a simple example using this pattern. The consumer counts the cars received. Each time it receives `1000` cars it prints a message on the display of the Browser.

You may run this program using:

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

When you feed a statement into the emulator, it is executed in its own thread. Therefore, after feeding the above statement two threads are created. The main one is for the consumer, and the forked thread is for the producer.

Notice that the consumer was written using the *recursive* pattern. Can we write this program using the iterative `ForAll/2` 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. We either introduce a stateful (mutable) data structure, which we will do in Section 9.4, or define another iterator that passes the state around. In our case, some iterators that fit our needs exist in the module `List`. First, we need an iterator that filters away all items except volvo's. We can use `{Filter Xs P ?Ys}` which outputs in `Ys` all the elements that satisfies the procedure `P/2` used as a Boolean function. The second construct is `{List.forAllInd Xs P}` which is similar to `ForAll`, but `P/2` takes the index of the current element of the list, starting from `1`, as its first argument, and the element of the list as its second argument. Here is the program:

Figure 8.7 Producing volvo's

```
fun {Producer N}
   if N > 0 then
      volvo|{Producer N-1}
   else nil end
end
local
   proc {ConsumerN Ls N}
      case Ls of nil then skip
      [] volvo|Lr then
         if N mod 1000 == 0 then
            {Browse 'riding a new volvo'}
         end
         {ConsumerN Lr N+1}
      else
         {ConsumerN {List.drop Ls 1} N}
      end
   end
in
   proc {Consumer Ls} {ConsumerN Ls 1} end
end
```

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

8.3 Thread Priority and Real Time

Try to run the program using the following statement:

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

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 has to do with the asynchronous message passing. If the producer sends messages i.e. create new elements in the stream, in a faster rate than the consumer can consume, increasingly more buffer-

ing will be needed until the system starts to break down.[3] There are a number of ways to solve this problem. One is to create a bounded buffer between producers and consumers which we will discuss later. Another way is to change the thread execution speed (by changing the thread's priority) so that consumers get more time-slices than producers.

The modules `Thread` and `Property` provide a number of operations pertinent to threads. Some of these are summarized in Figure 8.8.

Figure 8.8 Thread operations

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
`{Property.get priorities ?Pr }`	Gets system-priority ratios
`{Property.put priorities(high:+X medium:+Y)}`	Sets system-priority ratios

Oz has three priority levels. The system procedure

```
{Property.put 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

```
{Property.put priorities(high:10 medium:10)}
```

for each `10` time-slices allocated to runnable high-priority threads, 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. 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`.

```
local L in
   {Property.put threads priorities(high:10 medium:10)}
   thread
     {Thread.setThisPriority low}
      L = {Producer 5000000}
   end
   thread
```

---

[3]Ironically in the Mozart system, using the distributed programming capabilities, stream communication across sites works better because of designed flow control mechanism that suspends producers when the network buffers are full.

```
                {Thread.setThisPriority high}
                {Consumer L}
         end
   end
```

8.4 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 this case, constructs the stream with unbound variables (empty boxes). The producer waits for the unbound variables (empty boxes) to appear on the stream. It then binds the variables (fills the boxes). The general pattern of the producer is as 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 8.9 is a demand driven version of the program in Figure 8.7. You can run it with very large number of volvo's!

8.4.1 Futures

There is another way to program demand-driven computations. This uses the notion of *future* and the `ByNeed` primitive operation. A future is a read-only capability of a logic variable. For example to create a future of the variable `X` we perform the operation `!!` to create a future `Y`.

```
Y = !!X
```

A thread trying to use the value of a future, e.g. using `Y`, will suspend until the variable of the future, e.g. `X`, gets bound.

One way to execute a procedure lazily, i.e. in a demand-driven manner, is to use the operation `{ByNeed +P ?F}`. `ByNeed` takes a one-argument procedure `P`, and returns a future `F`. When a thread tries to access the value of `F`, the procedure `{P X}` is called, and its result value `X` is bound to `F`. This allows us to perform demand-driven computations in a straightforward manner. For example by feeding

Figure 8.9 Producing Volvo's lazily

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

```
declare Y
{ByNeed proc {$ X} X=1 end Y}
{Browse Y}
```

we will observe that `Y` becomes a future, i.e. we will see `Y<Future>` in the Browser. If we try to access the value of `Y`, it will get bound to `1`. One way to access `Y` is by performing the operation `{Wait Y}` which triggers the producing procedure.

Now we can rewrite program of Figure 8.9 as shown in Figure 8.10. This looks very similar to Figure 8.7

8.5   Thread Termination-Detection

We have seen how threads are forked using the statement **thread** *S* **end**. A natural question that arises is how to join back a forked thread 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 because Oz is a data-flow language.

```
thread T1 X1=unit end
thread T2 X2=X1 end
...
thread TN XN=XN-1 end
```

Figure 8.10 Producing Volvo's using `ByNeed`

```
local
   proc {Producer Xs}
      Xr in
      Xs = volvo|{ByNeed {Producer Xr} $}
   end
   proc {Consumer N Xs}
      if N>0 then
         case Xs of X|Xr then
            if X==volvo then
               if N mod 1000 == 0 then
                  {Browse 'riding a new volvo'}
               end
               {Consumer N-1 Xr}
            else {Consume N Xr} end
         end
      end
   end
in
   {Consumer 10000000 thread {Producer $} end}
end
```

$$\{\texttt{Wait}\ XN\}$$
$$MainThread$$

When All threads terminate the variables *X1* `...` *XN* will be merged together and bound to `unit`. {`Wait` *XN*} suspends the main thread until *XN* is bound.

In Figure 8.11 we define a higher-order construct (combinator), that implements the concurrent-composition control construct that has been outlined above. 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.

Figure 8.11 Concurrent Composition

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

The program Figure 8.5 didn't terminate properly when the `Ping` and the `Pong` threads terminated. This problem can be remedied now. If we use `Application.exit/1` a stand-alone application terminates aborting remaining threads. We can arrange things such that the main thread terminates only when the `Ping` and the `Pong` threads terminate. This is shown in Figure 8.12.

Figure 8.12 A 'Ping Pong' program stand-alone

```
functor
import
   Browser(browse:Browse) %Import Browse form Browser module
   Application
define
   proc {Ping N}
      if 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
   X1 X2
in
   {Browse 'game started'}
   thread {Ping 50} X1=unit end
   thread {Pong 50} X2=X1 end
   {Wait X2}
   {Application.exit 0}
end
```

Stateful Data Types

Oz provides a 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. For example, objects are implemented in a totally different way depending on the optimization level of the compiler. 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 exists. In addition, a member ceases to exist when it is no longer accessible.

9.1   Ports

Port is such an abstract data-type. A Port `P` is an asynchronous communication channel that can be shared among several senders. A port has a stream associated with it. The operation: `{Port.new S ?P}` creates a port `P` and initially connects it to the variable `S` taking the role of a stream. The operation: `{Port.send P M}` will append the message `M` to the end of the stream associated with `P`. 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. In order to protect the stream `S` from being bound by mistake `S` is actually a future. The following program shows a simple example using ports:

```
declare S P
P = {Port.new S}
{Browse S}

{Port.send P 1}
{Port.send P 2}
```

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

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

Ports are more expressive abstractions than pure stream communication, which was discussed in Section 8.2, since they can be shared among multiple threads, and can be embedded in other data structures. Ports are the main message passing mechanism between threads in Oz.

9.2   **Client-Server Communication**

Ports are used as a communication entry point to servers. The program shown in Figure 9.1 defines a thread that acts as FIFO queue server. It has two ports, one for inserting items to the queue using `put`, and the other for fetch items out of the queue using `get`. The use of single-assignment (logic) variables makes the server insensitive to the relative arrival order of `get` and `put` requests. `get` requests can arrive even when the queue is empty. A server is created by {NewQueueServer ?Q}. This procedure returns back a record `Q` with features `put` and `get` each holding a unary procedure. A client thread having access to `Q` can request services by invoking these procedure. Notice how results are returned back through logic variables. A client requesting an Item in the queue will call {Q.get I}. The server will eventually answer back by binding `I` to an item.

Figure 9.1 Concurrent Queue server, first attempt

```
declare
fun {NewQueueServer}
   Given GivePort={Port.new Given}
   Taken TakePort={Port.new Taken}
in
   Given = Taken
   queue(put:proc {$ X} {Port.send GivePort X} end
         get:proc {$ X} {Port.send TakePort X} end)
end
```

Try the following sequence of statements. The program will not work. So, what is the problem?

```
declare
thread Q = {NewQueueServer} end
{Q.put 1}
{Browse {Q.get $}}
{Browse {Q.get $}}
{Browse {Q.get $}}
{Q.put 2}
{Q.put 3}
```

The problem is that `Given = Taken` is trying to impose equality between two futures. Remember that `Given` and `Taken` are futures that can only be read and cannot be bound. So the thread corresponding to the queue server will suspend in the statement `Given = Taken`. This problem is remedied by running this statement in its own thread as shown in Figure 9.2 [1].

---

[1] This design of a FIFO queue server was proposed by Denys Duchier

The program works as follows. `{Q.put I0} {Q.put I1} ... {Q.put In}` will in-
crementally add the elements `I0 I1 ... In` to the stream `Given`, resulting in `I0|I1|...|In|<Future`
`{Q.get X0} {Q.put X1} ... {Q.put Xn}` will add the elements `X0 X1 ... Xn` to
the stream `Taken` resulting in `X0|X1|...|Xn|<Future2>`. The equality constraint
`Given = Taken` will bind `Xi's` to `Ii's`.

Figure 9.2 Concurrent Queue server

```
declare
fun {NewQueueServer}
   Given GivePort={Port.new Given}
   Taken TakePort={Port.new Taken}
in
   thread Given=Taken end
   queue(put:proc {$ X} {Port.send GivePort X} end
         get:proc {$ X} {Port.send TakePort X} end)
end
```

The code above is correct but, due to a limitation in the current Oz implementation,
leaks memory. As a workaround, one can use the code below as a drop-in replacement.

```
declare
fun {NewQueueServer}
   Given GivePort={Port.new Given}
   Taken TakePort={Port.new Taken}
in
   thread
      for X in Given Y in Taken do
         X=Y
      end
   end
   queue(put:proc {$ X} {Port.send GivePort X} end
         get:proc {$ X} {Port.send TakePort X} end)
end
```

9.3 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 an oz-name that is visible only (by lexical
scoping) to user-defined operations on the chunk.

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

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

This will display the following.

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

As a syntactic convenience, one can equate an expression E at an expression position with a variable X = E, and use X to refer to the value of the expression. Using this notation the above program could be written as

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

In Figure 9.4, we show an example of using the information hiding ability of chunks to implement Ports.

9.4 Cells

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 the initial content X. C is bound to a cell. The Figure 9.3 shows the operations on a cell.

Figure 9.3 Cell operations

Operation	Description
{NewCell X ?C}	Creates a cell C with content X
X=@C	Returns the content of C in X
C:=Y	Modifies the content of C to Y
{IsCell +C}	Tests if C is a cell
{Exchange +C X Y}	Swaps atomically the content of C from X to Y

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 @I}
   {Assign I b}    {Browse @I}
   {Assign I X}    {Browse @I}
   X = 5*5
   {Exchange I O thread O+1 end} {Browse @I}
end
```

Cells and higher-order iterators allow conventional assignment-based programming in Oz. The following program accumulates in the cell J the value of $\sum_{i=1}^{1} 0i$.

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

Ports described in Section 9.1 can be implemented by chunks and cells in a secure way, i.e. as an abstract data type that cannot be forged. The program in Figure 9.4 shows an implementation of Ports. Initially an Oz-name is created locally, which is accessible only by the Port operations. A port is created as a chunk that has one component, which is a cell. The cell is initialized to the stream associated with the port. The type test `IsPort` is done by checking the feature `Port`. Sending a message to a port results in updating the stream atomically, and updating the cell to point to the tail of the stream.

The implementation in Figure 9.4 does not protect the stream of the port. Protection of the stream is done using a future as follows.

Figure 9.4 Implementation of Ports by Cells and Chunks

```
declare Port in
local
   PortTag = {NewName} %New Oz name
   fun {NewPort S}
      C = {NewCell S} in
      {NewChunk port(PortTag:C)}
   end
   fun {IsPort ?P}
      {Chunk.hasFeature P PortTag} %Checks a chunk feature
   end
   proc {Send P M}
      Ms Mr in
      {Exchange P.PortTag Ms Mr}
      Ms = M|Mr
   end
in Port = port(new:NewPort
               is:IsPort
               send:Send)
end
```

Figure 9.5 Implementation of Ports by Cells and Chunks

```
declare Port in
local
   PortTag = {NewName} %New Oz name
   fun {NewPort FS}
      S C = {NewCell S} in
      FS = !!S  % Create a future
      {NewChunk port(PortTag:C)}
   end
   fun {IsPort ?P}
      {Chunk.hasFeature P PortTag} %Checks a chunk feature
   end
   proc {Send P M}
      Ms Mr in
      {Exchange P.PortTag Ms Mr}
      Ms = M|!!Mr
   end
in Port = port(new:NewPort
               is:IsPort
               send:Send)
end
```

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. Each attribute is a stateful cell that is accessed by the attribute-name, which is either an atom or an Oz-name.

- 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 either an atom or an Oz-name.

- Classes are stateless Oz-values[1]. Contrary to languages like Smalltalk, or Java etc., they are just descriptions of how the objects of the class should behave.

10.1 Classes from First Principles

Figure 10.1 shows how a class is constructed from first principles as outlined above. Here we construct a `Counter` class. It has a single attribute accessed by the atom val. It has a method table, which has three methods accessed through the chunk features `browse`, `init` and `inc`. A method is a procedure that takes a message, always a record, an extra parameter representing the state of the current object, and the object itself known internally as **self**.

As we can see, the method `init` assigns the attribute `val` the value `Value`, the method `inc` increments the attribute `val`, and the method `browse` browses the current value of `val`.

10.2 Objects from First Principles

Figure 10.2 shows a generic procedure that creates 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/2` that iterates over all fields of a record. `NewObject` returns

---

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

---

Figure 10.1 An Example of Class construction

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

---

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[2].

Figure 10.2 Object Construction

```
proc {NewObject Class InitialMethod ?Object}
   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
```

We can try our program as follows

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

---

[2]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

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 that the object application

```
{C inc(X)}
```

suspends inside the procedure `Inc/3` that implements method `inc`. 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}
```

10.3 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 application (calling a method in objects) is as cheap as procedure calls. The class `Counter` defined earlier has the syntactic form shown in Figure 10.3:

Figure 10.3 Counter Class

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

A class *X* is defined by:

```
class X ... end
```

Attributes are defined using the attribute-declaration part before the method-declaration part:

```
attr A1 ... AN
```

Then follows the method declarations, each has the form:

```
meth E S end
```

where the expression *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 statement *A* `:=` *E*.

A class can be defined anonymously by:

```
X = class $ ... end
```

The following shows how an object is created from a class using the procedure `New/3`, whose first argument is the class, the second is the initial method, and the result is the object. `New/3` 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
```

10.3.1 Static Method Calls

Given a class *C* and a method head *m*(`...`), a method call has the following form:

C, m(`...`)

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 denoted by `self` as implicit argument. The method could be defined at the class `C` or inherited from a super class. Inheritance will be explained shortly.

10.3.2 Classes as Modules

Static method calls have in general the same efficiency as procedure calls. This allows classes to be used as module-specification. This may be advantageous because classes can be built incrementally by inheritance. The program shown in Figure 10.4 shows a possible class acting as a module specification. 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 example of inheritance.

```
class ListC from BaseObject
```

We also show functional methods, i.e. methods that return results similar to functions. A functional method has in general the following form:

```
meth m( ... $) S E end
```

Here the class `ListC` inherits from the predefined class `BaseObject` that has only one trivial method: `meth noop() skip end`.

To create a module from the module specification one needs to create an object from the class. This is done by:

Figure 10.4 List Class

```
class ListC from BaseObject
   meth append(Xs Ys $)
      case Xs
      of nil then Ys
      [] X|Xr then
         X|(ListC , append(Xr Ys $))
      end
   end
   meth member(X L $)
      {Member X L}    % This defined in List.oz
   end
   meth length(Xs $)
      case Xs
      of nil then 0
      [] _|Xr then
        (ListC , length(Xr $)) + 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
```

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

ListM is an object that acts as a module, i.e. it encapsulates a group of procedures (methods). 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]  $)}}
```

10.4 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*, 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[3], and methods are available in the new class. We will restrict our discussion of inheritance to methods. Nonetheless, the same rules apply to features and attributes.

The methods available in a class *C* (i.e. visible) are defined through a precedence relation on the methods that appear in the class hierarchy. We call this relation the *overriding relation*:

- A method in a class *C* overrides any method, with the same label, in any super class of *C*.

Now a class hierarchy with the super-class relation can be seen as a directed graph with the class being defined as the root. The edges are directed towards the subclasses. 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
```

Figure 10.5 Illegal class hierarchy
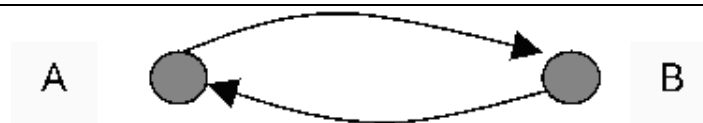
Second, after striking out all overridden methods each remaining method should have a unique label and is defined only in one class in the hierarchy. Hence, class C in the following example is not valid because the two methods labeled m remain.

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

Also the class C below is invalid, since two methods m is available in C.

```
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, and are completed by demand, using method caches, at execution time.

---

[3]To be defined shortly

Figure 10.6 Illegal class hierarchy in method m
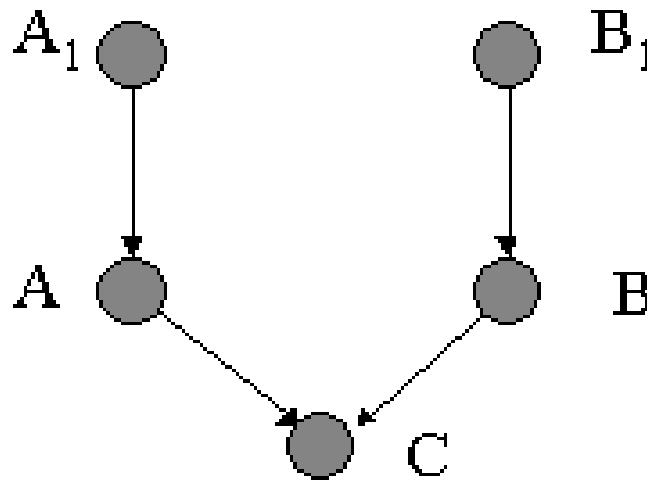
10.4.1 Multiple inheritance or Not

My opinion is the following:

- In general, to use multiple inheritance correctly, one has to understand the total inheritance hierarchy, which is sometimes worth the effort. This is important when there is a shared common ancestor.

- Oz restricts multiple inheritance in a way that most the problems with it do not occur.

- Oz enforces a programming methodology which requres one to override a method which is defined at more than one superclass, one has to define the method locally to 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 executing an initialization method in a class, one has to initialize its super classes. The only remedy here is to understand carefully the inheritance hierarchy to avoid such replication. Alternatively, you should only inherit from multiple classes that do not share stateful common ancestor. This problem is known as the implementation-sharing problem.

10.5 Features

Objects may have features similar to records. Features are stateless 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:

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

10.5.1 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 their corresponding values. Therefore, the following program will display `york` twice.

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

We may leave a feature uninitialized as in:

```
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, the following program will bind the feature `streetName` of object `Apt3` to the atom `kungsgatan`, and the corresponding feature of `Apt4` to the atom `sturegatan`.

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

One more form of initialization is available. A feature may be initialized in the class declaration to a variable or an Oz-value that has a variable. In the following, the feature is initialized to a tuple with an anonymous variable. In this case, all instances of the class will *share* the same variable. Consider the following program.

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

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

If entered incrementally, will show that the statement

```
Apt1.streetName = f(york)
```

binds the corresponding feature of `Apt2` to the same value as that of `Apt1`.

What has been said of features also holds for attributes.

10.6 Parameterized Classes

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

In Oz, we have also another natural method for creating generic classes. Since classes are first-class values, we can instead define a function that takes some type argument(s) and return a class that is specialized for the type(s). In Figure 10.7, the function `SortClass` is defined that takes a class as its single argument and returns a sorting class specialized for the argument.

We can now define two classes for integers and rationals:

```
class Int
   meth less(X Y $)
      X<Y
   end
end
class Rat from Object
   meth less(X Y $)
```

Figure 10.7 Parameterized Classes

```
fun {SortClass Type}
   class $ 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({self qsort(S $)} P|{self 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
            {self partition(Xr P Sr Lr)}
         end
      end
   end
end
```

```
                         '/'(P Q) = X
                         '/'(R S) = Y
                          in
                         P*S < Q*R
                     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)] $)}}
```

10.7   Self Application

The program in Figure 10.7 shows in the method `qsort` an object application using the keyword **self** (see below).

```
meth qsort(Xs Ys)
    case Xs
```

```
    ...
    {self partition(Xr P S L)}
    ...
end
```

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 (available) in the current object. Thereafter we apply the object (as a procedure) to the message. This is a form of dynamic binding common in all object-oriented languages.

10.8 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 using the keyword **attr** instead. When an object is created each attribute 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 and 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 10.8 we define the class Point. Note that the attributes x and y are initialized to zero before the initial message is applied. The method move uses **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)}
```

10.9 Private and Protected Methods

Methods may be labeled by variables instead of literals. These methods are *private* to the class in which 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:

Figure 10.8 The class Point

```
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
    % Switch the browser to virtual string mode
      {Browse "point at ("#@x#" , "#@y#")\n"}
   end
end
```

```
local A = {NewName} in
   class C from ...
      meth !A(X) ... end
      meth a(...) {self A(5)} ... end
      ...
   end
end
```

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 first making it private and second by storing it 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 create a subclass `C1` of `C` and access method `A` as follows:

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

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

Let us continue our simple example in Figure 10.8 by defining a specialization of the class that in addition of being a point, it stores a history of the previous movement. This is shown in Figure 10.9.

Figure 10.9 The class History Point

```
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  % made protected method
      {Browse "with location history: "}
      {Browse @history}
   end
end
```

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 `HistoryPoint` class. Second, `DisplayHistory` method is made private to the class. Moreover it is made available for subclasses, i.e. protected, by storing it in 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)}
```

10.10   Default Argument Values

A method head may have default argument values. Consider the following example.

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

A call of the method `m` may leave the arguments of features `d1` and `d2` unspecified. In this case these arguments will assume the value zero.

We continue our `Point` example by specializing `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 the area will be ignored. The class is shown in Figure 10.10. Notice that the method `init` has two default arguments that give a default area if not specified in the initialization of a new 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. We, any way, anticipated this problem and therefore created two protected methods stored in `boundConstraint` and `displayHistory` to avoid repeating the same actions. In any case, we have to refine the methods `init`, `move` and `display` since they occur in the two sibling classes. The solution is shown in Figure 10.11. 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 any harm. Try the following example:

```
declare P
P = {New BHPoint init(2 0)}
{P display}
{P move(1 2)}
```

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

Figure 10.10 The class BoundedPoint

```
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)
      if {self BoundConstraint(X Y $)} then
        Point,move(X Y)
      end
   end
   meth BoundConstraint(X Y $)
      (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 = "xbounds=("#X0#","#X1#"),ybounds=("
          #Y0#","#Y1#")"
   in
      {Browse S}
   end
end
```

Figure 10.11 The class BHPoint

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

Objects and Concurrency

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 an operation. In Oz, we separate the issue of acquiring exclusive access of an object from the object system. This gives us the ability to perform coarse-grain atomic operation on a set of objects, a very important requirement, e.g. in distributed database systems. The basic mechanism in Oz to get 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. A concurrent attempt to obtain the same lock will block until the thread currently holding it has released it.

11.1.1 Simple Locks

In the case of a simple lock, a nested attempt by the same thread to reacquire the same lock during the dynamic scope of a critical section guarded by the lock will block. We say *reentrancy* is not supported. Simple locks can be modeled in Oz as follows, where `Code` is a nullary procedure encapsulating the computation to be performed in the critical section. The lock is represented as a procedure: when applied to some code, it tries to get the lock by waiting until `Old` gets bound to `unit`. Notice that the lock is released upon normal as well as abnormal exit.

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

```
        {Exchange Cell Old New}
        {Wait Old} {Code}
     finally New=unit end
   end
end
```

11.1.2 Atomic Exchange on Object Attributes

Another implementation is using an object as shown below to implement a lock. Notice the use of the construct:

```
Old = lck := New
```

Similar to the Exchange operation on cells, this is an atomic exchange on an object attribute.

```
class SimpleLock
   attr lck:unit
   meth init skip end
   meth 'lock'(Code)
      Old New in
      try
         Old = lck := New
         {Wait Old} {Code}
      finally New= unit end
   end
end
```

11.2 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 acquired by at most one thread at a time. Concurrent threads that attempt to get 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 modeled in Oz as follows:

```
class ReentrantLock from SimpleLock
   attr Current:unit
   meth 'lock'(Code)
      ThisThread = {Thread.this} in
      if ThisThread == @Current then
         {Code}
      else
         proc {Code1}
            try
```

```
                                Current := ThisThread
                                {Code}
                            finally
                                Current := unit
                            end
                    end
                in
                    SimpleLock, 'lock'(Code1)
                end
            end
        end
```

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

```
lock E then S end
```

E is an expression that evaluates to a lock. The construct blocks until *S* is executed. If *E* is not a lock, then a type error is raised.

- {NewLock L} creates a new lock L.

- {IsLock E} returns true iff E is a lock.

11.2.1 Arrays

Oz has arrays as chunk subtype. Operations on arrays are defined in module Array.

- {NewArray +L +H +I ?A} creates an array A, where L is the lower-bound index, H is the higher-bound index, and I is the initial value of the array elements.

- {Array.low +A ?L} returns the lower index.

- {Array.high +A ?L} returns the higher index.

- R:=A.I returns A[I] in R.

- A.I:=X assigns X to the entry A[I].

As a simple illustration of the use of locks consider the program in Figure 11.1. The procedure Switch transforms negative elements of an array to positive, and zero elements to the atom zero! The procedure Zero resets all elements to zero.

Try the following program.

```
local X Y in
   thread {Zero A} X = unit end
   thread {Switch A} Y = X end
   {Wait Y}
   {For 1 10 1 proc {$ I} {Browse A.I} end}
end
```

Figure 11.1 Using a lock

```
declare A L in
A = {NewArray 1 100 ~5}
L = {NewLock}
proc {Switch A}
   {For {Array.low A} {Array.high A} 1
    proc {$ I}
       X := A.I in
       if X<0 then A.I := ~X
       elseif X == 0  then A.I := zero end
       {Delay 100}
     end}
end
proc {Zero A}
   {For {Array.low A} {Array.high A} 1
    proc {$ I} A.I := 0 {Delay 100} end}
end
```

The elements of the array will be mixed `0` and `zero`.

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  A.I} end}
end
```

By Switching the delay statement above between the first and the second thread, 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 using multiple locks.

11.3   Locking Objects

To guarantee mutual exclusion on objects one may use the locks described in the previous subsection. Alternatively, we may declare in the class that its instance objects

can be locked with a default lock existing in the objects when they are created. A class with an implicit lock is declared as follows:

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

This does not automatically lock the object when one of its methods is called. Instead we have to use the construct:

```
lock S end
```

inside any 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 have 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 calling methods in multiple objects, we might deadlock if there is any cyclic dependency. Writing nontrivial concurrent programs needs careful understanding of the dependency 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 10.3 can be refined to work in concurrent environment by refining it as follows:

```
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.

11.4 Concurrent FIFO Channel

The first example shows a concurrent channel, which is shared among an arbitrary number of threads. Any 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 11.2 shows one possible realization. This program relies on the use of logical variables to achieve the desired synchronization. The method `put/1` inserts an element in the channel. A thread executing the method `get/1` will wait until an element is put in the channel. Multiple consuming threads will 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.

Figure 11.2 An Asynchronous Channel Class

```
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
```

11.5 Monitors

The next example shows a traditional way to write *monitors*. We start by defining a class that defines the notion of events and the monitor operations `notify(Event)` and `wait(Event)` by specializing the class `Channel`.

```
class Event from Channel
   meth wait
      Channel , get(_)
   end
   meth notify
      Channel , put(unit)
   end
end
```

We show here an example of a unit buffer in the traditional monitor style. The unit buffer behaves in a way very similar to a channel when it comes to consumers. Each consumer waits until the buffer is full. In the case of producers only one is allowed to insert an item in the empty buffer. Other producers have to suspend until the item is consumed. The program in Figure 11.3 shows a single buffer monitor. Here we had to program a signaling mechanism for producers and consumers. Observe the pattern in `put/1` and `get/1` methods. Most execution is done in an exclusive region. If waiting is necessary it is done outside the exclusive region. This is done by using an auxiliary variable `X`, which gets bound to `yes`. The `get/1` method notifies one producer at a time by setting the `empty` flag and notifying one producer (if any). This is done as an atomic step. The `put/1` method does the reciprocal action.

Try the above example by running the following code:

```
local
  UB = {New UnitBufferM init} in
  {For 1 15 1
   proc{$ I} thread {UB put(I)} {Delay 500} end end}
  {For 1 15 1
   proc{$ I} thread {UB get({Browse}}{Delay 500} end end}
end
```

11.5.1 Bounded Buffers Oz Style

In Oz, it is very rare to write programs in the monitor style shown above. In general it is very awkward. There is a simpler way to write a `UnitBuffer` class that is not traditional. This is due to the combination of objects and logic variable, Figure 11.4 shows a simple definition. No locking is needed directly.

A simple generalization of the above program leads to an arbitrary size bounded buffer class. This is shown in below. The put and get methods are the same as before. Only the initialization method is changed.

```
class BoundedBuffer from UnitBuffer
   attr prodq buffer
   meth init(N)
      buffer := {New Channel init}
      prodq := {New Event init}
      {For 1 N 1 proc {$ _} {@prodq notify} end}
   end
end
```

11.6 Active Objects

An active object is a thread whose behavior is described by a class definition. Communication with active objects is through asynchronous message passing. An active object reacts to received messages by executing the corresponding methods in its associated class. An active object executes one method at a time. Therefore locking is not needed for methods performed by an active object. The interface to an active object is through

Figure 11.3 A Unit Buffer Monitor

```
class UnitBufferM
   attr item empty psignal csignal
   prop locking
   meth init
      empty := true
      psignal := {New Event init}
      csignal := {New Event init}
   end
   meth put(I)
      X in
      lock
         if @empty then
            item := I
            empty := false
            X = yes
            {@csignal notify}
         else X = no end
      end
      if X == no then
         {@psignal wait}
         {self put(I)}
      end
   end
   meth get(I)
      X in
      lock
         if {Not @empty} then
            I = @item
            empty := true
            {@psignal notify}
            X = yes
         else X = no end
      end
      if X == no then
         {@csignal wait}
         {self get(I)}
      end
   end
end
```

Figure 11.4 Unit Buffer

```
class UnitBuffer from BaseObject
   attr prodq buffer
   meth init
      buffer := {New Channel init}
      prodq := {New Event init}
      {@prodq notify}
   end
   meth put(I)
      {@prodq wait}
      {@buffer put(I)}
   end
   meth get(?I)
      {@buffer get(I)}
      {@prodq notify}
   end
end
```

Oz ports. Clients of an active object send messages to the object by sending messages
to its associated port. We will show how to create generically this abstraction. Since
active objects resemble servers receiving messages from clients though a network we
call this abstraction the server abstraction. To create a server S from a class Class we
execute:

```
S = {NewServer Class init}
```

Here init is the initial object construction method. To get the basic idea we show first
a simplified form of the NewServer function. The following function:

- creates a port Port,

- creates an object Object, and finally

- creates a thread that serves messages sent to the port, by applying the corre-
  sponding class methods.

```
fun {NewServer Class Init}
   S    % The stream of the port
   Port = {NewPort S}
   Object = {New Class Init}
in
   thread {ForAll S
            proc{$ M} {Object M} end}
   end
   Port
end
```

We would like to add the ability of terminating the thread by making a protected method `Close` accessible to the method in `Class`. This leads us to the following extension of the above function. We use the exception handling mechanism to jump out of the receiving loop.

```
local
   class Server
     attr close:Close
     meth Close raise closeException end end
   end
in
fun {NewServer Class Init}
   S     % The stream of the port
   Port = {NewPort S}
   Object = {New class $ from Server Class end Init}
in
   thread
     try  {ForAll S
             proc{$ M} {Object M} end}
     catch closeException then skip end
   end
   Port
end
```

Logic Programming

Many problems, especially frequent in the field of Artificial Intelligence, and also found elsewhere, e.g., in operations research, are currently solvable only by resorting to some form of search and *constraint* propagation. Such problems can be specified very concisely, if the programming language abstracts away the details of search by providing don't know nondeterminism. Logic programming and Prolog is considered a suitable formalism for this class of problems. In this chapter we will talk about how to express logic programming and concurrent constraint programming in Oz. In logic programming each procedure can be interpreted as a relation expressed by a logical statement. We will also discuss the relation between Oz and Prolog, and how most Prolog programs have a straight forward translation to Oz programs. For more advanced constraint solving techniques, the reader may look to the companion tutorial on constraint programming in Oz.

12.1 Constraint Stores

Oz threads share a store where variable bindings are stored in the form of equalities: $X_1 = U_1, \ldots, X_n = U_n$ where X_i are variables and U_i are either Oz entities or variables. The constraint store contains the Oz values corresponding to records, numbers, names or variables, and the names that uniquely idententifies the procedures, cells and the various types of chunks (classes, objects, functors, etc.). Conceptually the store is modeled as a conjunctive logical formula: $\exists Y_1 \ldots Y_m : X_1 = U_1 \wedge \ldots \wedge X_n = U_n$, where the X_i are the variables and U_i are Oz values or variables, and Y_i are the union of all variables occuring in X_i and U_i. The store is said to be a *constraint store*. An Oz *computation store* consists of a constraint store, a procedure store where procedures reside, and a cell store where cells and object states reside.

12.2 Computation Spaces

A computation space consists in general of a computation store and a set of executing threads. What we have seen so far is a single computation space. When dealing with logic programming a more elaborate structure will arise with multiple nested computation spaces. The general rules for the structure of computation spaces are as follows.

- There is always a topmost computation space where threads may interact with the external world. A thread trying to add inconsistent constraints (bindings) to

the store of the top space will raise a failure exception in the thread. The addition of the inconsistent constraints will be aborted and the constraint store remains always consistent.

- A thread may create a local computation space either directly or indirectly as will be shown in this section. The new computation space will be a child space and the current one the parent space. In general a hierarchy of computation spaces may be created.

- A thread belongs always to one computation space. Also, variables belong to only one computation space.

- A thread in a child space sees and may access variables belonging to its space as well as to all ancestor spaces. The converse is false. A thread in a parent space cannot see the variables of a child space, unless the child space is merged with the parent. In such a case, the child space disappears, and all its content is added to the parent space. The space merge operation may occur due to an explicit operation, or indirectly due to a language construct as will be seen in this section.

- A Thread in a child space my add constraints (bindings) on variables visible to it. This means that it may bind variables belonging to its space or to its ancestor spaces. The binding will only be visible in the current space and all its children spaces if any.

12.3 Constraint Entailment and Disentailment

A condition C is entailed by the store σ if C, considered as a logical formula, is logically implied by the store σ, again considered as a logical formula. Intuitively entailment of C means that adding C to the store does not increase the information already there. Everything is already there.

A condition C is disentailed by the store if the negation of C is logically implied by the store σ. A disentailed constraint is inconsistent with the information already in the store.

Since a constraint store is a logical formula, we can also talk of a constraint store being entailed, or disentailed by another constraint store. A space S_0 is entailed (disentailed) by another space S_1 if the constraint store of S_0 is entailed (disentailed) by the constraint store if S_1.

We call a space that is disentailed (normally by a parent space) a *failed space* .

12.3.1 Examples

Consider the store $\sigma \equiv X = 1 \wedge \ldots \wedge Y = f(X\ Z)$ and the following conditions:

- $X = 1$ is entailed since adding this binding does not increase the information in the store.

- $\exists U : Y = f(1\ U)$ is also entailed. Adding this information does not increase our information. There is a Z that satisfies the above condition. Notice that we do not know which value Z will assume. But whatever value assumed by Z, the condition would be still satisfied.

- $Y = f(1\ 2)$ is not entailed by the store, since adding this equality increases the information there, namely by making $Z = 2$.

- $X = 2$ or $Y = f(3\ U)$ are both disentailed since they contradict information already present. They will cause a failure exception to be raised: in the top space this is normally reported to the user in an error message, whereas a subordinated space is merely failed.

12.4 Disjunctions

Now we are in a position to understand the nondeterminate constructs of Oz. Oz provides several disjunctive constructs for *nondeterminate choice*, also known as *don't know choice* statements.

12.4.1 or statement

In all the disjunctive statements we are going to use the notion of a clause and a guard. A clause consists of a guard *G* and a body *S1*, and has the following form:

> *G* **then** *S1*

The guard *G* has the form:

> *X1* **...** *Xn* **in** *S0*

where the variables *Xi* are existentially quantified with scope extending over both the guard and the body.

The first disjunctive statement has the following form:

```
or
   G1 then S1
[] G2 then S2
   ...
[] GN then SN
end
```

An **or**-statement has the following semantics. Assume a thread is executing the statement in space *SP*.

- The thread is blocked.

- N spaces are created $SP_1, \ldots, SP_N$ with N new threads executing the guards $G_1, \ldots, G_N$.

- Execution of the father thread remains blocked until at most one of the child spaces is not failed.

- If all children spaces are failed, the parent thread raises a failure condition in its space. This means that if the space of the parent thread is the top space, a failure exception is raised. Otherwise the space is local and it becomes a failed space.

- Only one space remains that is not failed which corresponds to the clause G_i **then** S_i. Assume also that G_i has been reduced to the goal G'_i and the constraint θ. In this case, the space is merged with the parent space. θ and the variables of the store are added to that of the parent store. G'_i executes in its own thread, and the original suspending thread resumes executing the statement S_i. This rule of execution is called *unit commit* in Oz because execution commits to one alternative disjunct (the only one that is left).

12.4.2 Shorthand Notation

```
or
...
[] Gi
...
end
```

Stands for

```
or
...
[] Gi then skip
...
end
```

Observe that the **or** statement does not introduce any don't know nondeterminism. A thread executing such a statement waits until things works out in a determinate course of action.

12.4.3 Prolog Comparison

The **or** statement just described does not have a corresponding construct in Prolog. The Prolog disjunct `P ; Q` always creates a choice point that is subject to backtracking.

12.5 Determinacy Driven Execution

The **or**-statement of Oz allows a pure logical form of programming style where computations are synchronized by determinacy conditions. Consider the following program.

```
proc {Ints N Xs}
   or N = 0 Xs = nil
   [] Xr in
```

```
            N > 0 = true Xs = N|Xr
            {Ints N-1 Xr}
         end
      end
   local
      proc {Sum3 Xs N R}
         or Xs = nil R = N
         [] X|Xr = Xs in
            {Sum3 Xr X+N R}
         end
      end
   in proc {Sum Xs R} {Sum3 Xs 0 R} end
   end
   local N S R in
      thread {Ints N S} end
      thread {Sum S {Browse}} end
      N = 1000
   end
```

The thread executing `Ints` will suspend until `N` is known, because it cannot decide on which disjunct to take. Similarly, `Sum3` will wait until the list `S` is known. `S` will be defined incrementally and that will lead to the suspension and resumption of `Sum3`. Things will start to take off when the main thread binds `N` to `1000`. This shows clearly that determinacy driven execution gives the synchronization information need to mimic producer/consumer behavior.

12.6 Conditionals

12.6.1 Logical Conditional

A logical conditional is a statement having the following form.

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

where *Xi* are newly introduced variables, and *Si* are statements. *X1* `...` *XN* `in` *S0* `then` *S1* is the clause of the conditional, and *S2* is the alternative.

A `cond`-statement has the following semantics. Assume a thread is executing the statement in space *SP*.

- The thread is blocked.

- A space SP_1 is created, with a single thread executing the guard `cond` *X1* `...` *XN* `in` *S0*.

- Execution of the father thread remains blocked until SP_1 is either entailed or disentailed. Notice that these conditions may never occur, e.g. when some thread is suspending or running forever in SP_1.

- If SP_1 is disentailed, the father thread continues with *S2*.

- If SP_1 is entailed, assume it has been reduced to the store θ and the set of local variables SX In this case, the space is merged with the parent space. θ and SX added to the parent store, and the father thread continues with the execution of *S1*.

12.6.2 Prolog Comparison

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

12.6.3 Parallel Conditional

A parallel conditional is of the form

```
cond G1 then S1
[] G2 then S2
...
else SN end
```

A parallel conditional is executed by evaluating all conditions *G1* `...` *G(N-1)* in an *arbitrary* order, possibly concurrently, each in its own space. If one of the spaces, say *Gi*, is entailed, its corresponding statement *Si* is chosen by the father thread. If all spaces are failed, the else statement *SN* is chosen, otherwise the executing thread suspends.

Parallel conditionals are useful mostly in concurrent programming, e.g. for programming time-out on certain events. This construct is the basic construct in concurrent logic programming languages (also known as committed-choice languages).

As a typical example from concurrent logic programming let us define the indeterministic binary merge, where the arrival timing of elements on the two streams Xs and Ys determines the order of elements on the resulting stream Zs.

```
proc {Merge Xs Ys Zs}
   cond
      Xs = nil then Zs = Ys
   [] Ys = nil then Zs = Xr
   [] X Xr in Xs = X|Xr then Zr in
      Zs = X|Zr {Merge Xr Ys Zr}
   [] Y Yr in Ys = Y|Yr then Zr in
      Zs = Y|Zr {Merge Xs Yr Zr}
   end
end
```

In general binary-stream merge is inefficient, specially when multiple of these are used to connect multiple threads to a simple server thread. An efficient way to implement

a constant-time multi-merge operator is defined below by using cells and streams in-stread. The procedure {MMerge STs L} has two arguments STs may be either nil, a list of streams to merged, or of the form merge(ST1 ST2) where each STi is again of the same form as STs.

```
proc {MMerge STs L}
   C = {NewCell L}
   proc {MM STs S E}
      case STs
      of ST|STr then M in
         thread
            {ForAll ST proc{$ X} ST1 in {Exchange C X|ST1 ST1} end}
            M=S
         end
         {MM STr M E}
      [] nil then skip
      [] merge(STs1 STs2) then M in
         thread {MM STs1 S M} end
         {MM STs2 M E}
      end
   end
   E
in
   thread {MM  STs unit E} end
   thread if E==unit then L = nil end end
end
```

A binary-merge {Merge X Y Z} is simply {MMerge [X Y] Z}.

12.7 Nondeterministic Programs and Search

Oz allows much of the nondeterministic and search-oriented programming as Prolog. This type of programming comes in a little bit different flavour than Prolog. While Prolog comes ready with a default search strategy based on backtracking, Oz allows programmers to devise their suitable search strategies in a way that is separate and orthogonal from the nondeterministic specification of a problem.

To be able to do this Oz has a specific linguistic constructs that create choice point without specifying how they will be explored. A completely separate program can then specify the search strategy.

12.7.1 dis Construct

The following program uses the **dis** construct of Oz to create a choice point when necessary.

```
proc {Append Xs Ys Zs}
   dis
```

```
            Xs = nil Ys = Zs then skip
      [] X Xr Zr in
            Xs = X|Xr Zs = X|Zr then
            {Append Xr Ys Zr}
      end
end
```

It corresponds roughly to the `append/3` program of Prolog:

```
append([], Ys, Ys).
append([X|Xr], Ys, [X|Zr]) :- append(Xr, Yr, Zr).
```

In fact the same kind of abbreviations that hold for **or** hold also for **dis**. That is the above program have the following abbreviated form.

```
proc {Append Xs Ys Zs}
   dis
      Xs = nil Ys = Zs
   [] X Xr Zr in
      Xs = X|Xr Zs = X|Zr then
      {Append Xr Ys Zr}
   end
end
```

Assume the following procedure call:

```
local X in
   {Append [1 2 3] [a b c] X}
   {Browse X}
end
```

This will behave exactly as the **or** construct, i.e. it will deterministically bind `X` to `[1 2 3 a b c]`. If we on the other hand try:

```
local X Y in
   {Append X Y [1 2 3 a b c]}
   {Browse X#Y}
end
```

the bahavior will look the same as with the **or** construct; the thread executing this sequence of calls will suspend while executing `{Append X Y [1 2 3 a b c]}`. There is however a difference. The call of `Append` will create a choice-point with two alternatives:

- `X = nil Y = [1 2 3 a b c]` **then skip**

-     `Xr Xr` **in**
  `X = 1|Xr Zr = [2 3 a b c]` **then**
  `{Append Xr Y Zr}`

12.7.2 Define Clause Grammer

```
Sentence(P) --> NounPhrase(X P1 P) VerbPhrase(X P1)
NounPhrase(X P1 P) --> Determiner(X P2 P1 P) Noun(X P3) RelClause(X P3 P2)
NounPhrase(X P P) --> Name(X)
VerbPhrase(X P) --> TransVerb(X Y P1) NounPhrase(Y P1 P) | InstransVerb(X P
RelClause(X P1 and(P1 P2)) --> [that] VerbPhrase(X P2)
RelClause(_ P P) --> []
Determiner(X P1 P2 all(X imp(P1 P2))) --> [every]
Determiner(X P1 P2 exits(X and(P1 P2))) --> [a]
Noun(X man(X)) --> [man]
Noun(X woman(X)) --> [woman]
name(john) --> [john]
name(jan) --> [jan]
TransVerb(X Y loves(X Y)) --> [loves]
IntransVerb(X lives(X))  --> [lives]


proc {Sentence P S0#S}
   X P1 S1 in
   {NounPhrase X P1 P S0#S1}
   {VerbPhrase X P1 S1#S}
end
proc {NounPhrase X P1 P S0#S}
   choice
      P2 P3 S1 S2 in
      {Determiner X P2 P1 P S0#S1}
      {Noun X P3 S1#S2}
      {RelClause X P3 P2 S2#S}
   [] {Name X S0#S}
      P1 = P
   end
end
proc {VerbPhrase X P S0#S}
   choice
      Y P1 S1 in
      {TransVerb X Y P1 S0#S1}
      {NounPhrase Y P1 P S1#S}
   [] {IntransVerb X P S0#S}
   end
end
proc {TransVerb X Y Z S0#S}
   S0 = loves|S
   Z = loves(X Y)
end
proc {IntransVerb X Y S0#S}
   S0 = lives|S
   Y = lives(X)
end
proc {Name X S0#S}
```

```
              S0 = X|S
              choice
                 X = john
              []
                 X = jan
              end
           end
           proc {Noun X Y S0#S}
              choice
                 S0 = man|S
                 Y = man(X)
              [] S0 = woman|S
                 Y = woman(X)
              end
           end
           proc {Determiner X P1 P2 P S0#S}
              choice
                 S0 = every|S
                 P = all(X imp(P1 P2))
              [] S0 = a|S
                 P = exists(X and(P1 P2))
              end
           end
           proc {RelClause X P1 P S0#S}
              P2 in
              choice
                 S1 in
                 S0 = that|S1
                 P = and(P1 P2)
                 {VerbPhrase X P2 S1#S}
              [] S0 = S
                 P = P1
              end
           end

           declare
           proc {Main P}
              {Sentence P [every man that lives loves a woman]#nil}
           end
```

12.7.3 Some Search Procedures

12.7.4 Dis Construct

```
           declare Edge
           proc {Connected X Y}
              dis
                 {Edge X Y}
              [] Z in {Edge X Z}  {Connected Z Y}
```

```
            end
      end
      proc {Edge X Y}
         dis
              X = 1 Y = 2
         [ ] X = 2 Y = 1
         [ ] X = 2 Y = 3
         [ ] X = 3 Y = 4
         [ ] X = 2 Y = 5
         [ ] X = 5 Y = 6
         [ ] X = 4 Y = 6
         [ ] X = 6 Y = 7
         [ ] X = 6 Y = 8
         [ ] X = 1 Y = 5
         [ ] X = 5 Y = 1
         end
      end

      {ExploreOne
       proc {$ L}
          X Y in
          X#Y = L {Connected X Y}
       end
      }
      {Browse
       {SearchAll
        proc {$ L}
           X Y in
           X#Y = L {Connected X Y}
        end
       }}
```

12.7.5 Negation

```
      proc {NotP P}
         {SearchOne proc {$ L} {P} L=unit end $} = nil
      end


      proc {ConnectedEnh X Y Visited}
         dis
            {Edge X Y}
         [ ] Z in
            {Edge X Z}
            {NotP proc{$} {Member Z Visited} end}
            {ConnectedEnh Z Y Z|Visited}
         end
      end
```

12.7.6 Dynamic Predicates

```
proc {DisMember X Ys}
   dis Ys = X|_ [] Yr in Ys = _|Yr {DisMember X Yr} end
end


class DataBase from BaseObject
   attr d
   meth init
      d := {NewDictionary}
   end
   meth dic($) @d end
   meth tell(I)
      case {IsFree I.1} then
         raise database(nonground(I)) end
      else
         Is = {Dictionary.condGet @d I.1 nil} in
         {Dictionary.put @d I.1 {Append Is [I]}}
      end
   end
   meth ask(I)
      case {IsFree I} orelse {IsFree I.1} then
         {DisMember I {Flatten {Dictionary.items @d}}}
      else
         {DisMember I {Dictionary.condGet @d I.1 nil}}
      end
   end
   meth entries($)
      {Dictionary.entries @d}
   end
end


declare
proc {Dynamic ?Pred}
   Pred = {New DataBase init}
end
proc {Assert P I}
   {P tell(I)}
end
proc {Query P I}
   {P ask(I)}
end


EdgeP = {Dynamic}
{ForAll
[edge(1 2)
 edge(2 1)   % Cycle
 edge(2 3)
 edge(3 4)
```

```
       edge(2 5)
       edge(5 6)
       edge(4 6)
       edge(6 7)
       edge(6 8)
       edge(1 5)
       edge(5 1)  % Cycle
      ]
     proc {$ I} {Assert EdgeP I} end
     }
```

12.7.7 The Basic Space Library

12.7.8 Example: A Simple Expert System

Bibliography

[1] Denys Duchier. *Not Known*. DFKI Oz documentation series, 1998.

[2] Seif Haridi. *Not Known*. Not known, 1998.

[3] Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, 1994.

[4] Christian Shulte. *Not Known*. DFKI Oz documentation series, 1998.

[5] Gert Smolka. *Not Known*. DFKI Oz documentation series, 1995.

[6] Richard M. Stallman. *GNU Emacs Manual*, 7th edition, 1991.

[7] Peter Van Roy. *Not Known*. Not known, 1997.

Index

merge
    merge, merging nodes, 19

nesting
    nesting, marker, 30
nesting, 30

tell
    tell, incremental, 19

unification, 19