Application Programming

Denys Duchier
Leif Kornstaedt
Christian Schulte

Version 1.3.2
June 15, 2006

Abstract

This document is an introduction to application programming with Oz and Mozart.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

Part I

Getting Started

1

Getting Started

The purpose of programming languages is of course the construction of applications. In this chapter we will use Oz for our first small application.

1.1 Our First Application: `Webget.oza`

Our first application is a program that copies a file from a url to a local file. The program is called `Webget.oza`.

Figure 1.1 Get yourself a Mozart license.

```
ozengine Webget.oza --in http://www.mozart-oz.org/LICENSE --out LICENSE
```

Our goal is to have an application that can be started like any other application from the operating system shell. For example, executing the command shown in Figure 1.1 gets us a local copy of the Mozart license file.

In addition to taking arguments from the command line, `Webget.oza` should report problems by returning an error status to the operating system shell. The error status is an integer, where the value `0` means okay. Any other values signals an error.

1.2 What to do?

In the following we consider the three main steps in constructing our application. We give a brief outline of what to do, the following sections are going to detail the steps.

Definition The first step, of course, is to program our application. For this purpose, we will create a file `Webget.oz` that contains the Oz program implementing `webget`. More specifically, the file `Webget.oz` contains a functor definition.

Compilation Next, we compile the functor definition contained in `Webget.oz`. The compiler takes the functor definition and executes it. By this it creates a functor. Then the functor is written to a file `Webget.oza`. This persistent representation of a functor value is called a pickled functor.

Execution The pickled functor `Webget.oza` is executed by the Oz virtual machine `ozengine`. The engine takes a pickled functor (`Webget.oza` in our case), unpickles the functor, runs the functor, and supplies it with application arguments. After execution terminates, it reports the application's execution status back to the operating system shell.

1.3 Functor Definition: `Webget.oza`

The toplevel structure of the file `Webget.oz` is as follows.

4a ⟨**Webget.oz** 4a⟩≡
```
functor
   ⟨Module import 4b⟩
   ⟨Functor body 5a⟩
end
```

Importing modules Our application requires the system module `Application` to both process the command line arguments as well as for terminating the application. In addition, the module `Open` provides the class `Open.file` required for reading and writing files.

The functor needs to import these two modules. The functor definition thus contains the following import specification:

4b ⟨**Module import** 4b⟩≡
```
import
   Application
   Open
```

The import specification serves two purposes: the variable identifiers `Application` and `Open` are introduced with a lexical scope that extends over the entire body of the functor. Secondly, the identifiers also serve as module names. When the functor corresponding to this definition is executed, the variables are given as values the system modules with the same names.

More precisely, the import specification above is a convenient abbreviation for the more verbose specification below:

4c ⟨**Module import (no convenience)** 4c⟩≡
```
import
   Application at 'x-oz://system/Application'
   Open        at 'x-oz://system/Open'
```

In Section 3.3 we will discuss system modules in more detail. In particular, Figure 3.1 lists available system modules.

Functor body The body of a functor is a statement that is executed when the application is run.

5a ⟨Functor body 5a⟩≡
```
define
   ⟨Argument processing 5b⟩
   Status = try
               ⟨Opening input and output files 5c⟩
            in
               ⟨Copying input file to output file 6a⟩
               0
            catch _ then 1
            end
   ⟨Terminating the application 6b⟩
```

The structure for our application is straightforward: after processing the command line arguments, file objects for source and destination are created and the content is copied from the source file to the destination file.

If a runtime error occurs either during opening the files or while copying the file content, the raised exception is caught and the `Status` is bound to `1`. Otherwise, `Status` gets bound to zero.

Note that the body of a functor is like the part of a `local ... in ... end` statement before the `in`: definitions and statements are allowed, where the left hand side of definitions can introduce variables.

Processing arguments The application needs two input parameters: the URL to get the file from, and the file name under which the downloaded content should be stored.

The following application of `Application.getCmdArgs`

5b ⟨Argument processing 5b⟩≡
```
Args = {Application.getArgs record('in'(single type:string)
                                   'out'(single type:string))}
```

computes `Args` to be a record (as signalled by the label `record` of the single argument to `Application.getArgs`[1]. The features of the record are `'in'` and `'out'` where both fields are of type string and both are allowed to be given only once on the command line (that is specified by `single`).

For a complete reference of how application arguments are processed see Chapter *Application Support:* `Application`, *(System Modules)*.

Opening input & output The two files are opened as follows:

5c ⟨Opening input and output files 5c⟩≡

[1] Section *The* `Application` *Module, (System Modules)*

```
I={New Open.file init(url:  Args.'in')}
O={New Open.file init(name: Args.'out'
                      flags:[write create truncate])}
```

Note how the strings `Args.'in'` and `Args.'out'` computed by argument processing are used for the source URL and the destination filename.

Copying input to output Copying the file from source to destination is straightforward: As long as we can read a non-empty string `S` from the source file, we write it to the destination file and repeat the procedure.

6a ⟨**Copying input file to output file** 6a⟩≡

```
local
   proc {Copy}
      S={I read(list:$)}
   in
      if S\="" then
         {O write(vs:S)} {Copy}
      end
   end
in
   {Copy}
end
```

Terminating the application Termination of the application is effected by invocation of `Application.exit` which takes the application status as single integer argument. In our case an exit value of `1` indicates an error, otherwise `0` is returned to the operating system shell.

6b ⟨**Terminating the application** 6b⟩≡

```
{Application.exit Status}
```

1.4 Compilation

As is the case for many programming languages, the functor definition must be compiled before it can be executed. This is achieved by invoking the Oz compiler `ozc` as follows:

```
ozc -c Webget.oz -o Webget.oza
```

Note the intentional similarity between the options illustrated above and those accepted by a C compiler. The compiler `ozc` offers a variety of other options that control compilation, an overview of which can be found in Chapter *The Oz Compiler:* `ozc`, *(Oz Shell Utilities)*.

1.5 Execution

The functor pickled into `Webget.oza` can be executed by applying the program `ozengine` to the functor and the application arguments. For example, to copy the Mozart license file at url `http://www.mozart-oz.org/LICENSE` to the local file `LICENSE`, simply enter the command line shown in Figure 1.1 at your shell promt.

Execution of an application proceeds as follows:

1. `ozengine`, the Oz virtual machine, is started by the operating system.
2. `ozengine` starts to execute a module manager.
3. The module manager loads the pickled functor `Webget.oza`. This initial application functor is called the root functor.
4. The module manager links the functor by applying the functor body to argument modules. The argument modules in our example are the system modules `Application` and `Open`.
5. Then it executes the functor body.

The different steps in application are detailed in the following sections.

2

Application Development

Chapter 1 used a rather simple application as example. This chapter shows how to use functors for the modular development of larger applications.

2.1 Functors for Modular Applications

Principles of good software engineering suggest that larger applications should be designed and assembled from a collection of smaller modules. In Oz, this decomposition can be realized in terms of several functor definitions.

The primary purpose of a functor is to compute a module: It takes modules as input and computes a new module as output. As we have seen already, the `import` section of a functor specifies its inputs as a list of module names. In addition, functors may also have an `export` section which is basically a list of feature/value pairs that describes the module computed by the functor.

As demonstrated in Section 1.5 an application is run by executing the root functor. In our particular example, the root functor was rather simple in that it only imported system modules. However, larger applications will typically import modules computed by other application functors.

2.2 Example: Last Minute Flights

In the following we will build a trivial flight booking system featuring three components:

1. A data base server: It maintains a data base that contains available flights, where each flight has a unique id by which it can be identified. At first, the data base is not even persistent, but as we incrementally refine and improve our application, the data base evolves into a persistent and distributed data base server.
2. A graphical flight booking form, where a travel-minded user can choose a flight, enter her name, her E-mail address and so on. Later we will show how to build a web-based interface serving the same purpose.
3. The main component of our application that manages user requests to the data base and sets up the application.

All components are programmed as functors.

2.3 The Data Base

Let us start with the data base, which is the most straightforward part of our application. The data will be held in a dictionary that uses integers as keys, and arbitrary data structures as entries. The functor definition resides in file `DB.oz` and its toplevel structure is as follows:

10a ⟨**DB.oz** 10a⟩≡
```
functor
   ⟨Export specification for DB.oz 10b⟩
   ⟨Body for DB.oz 79a⟩
end
```

The functor has no import specification, and its export specification is as follows:

10b ⟨**Export specification for DB.oz** 10b⟩≡
```
export
   add:    Add
   get:    Get
   getAll: GetAll
   remove: Remove
```

The specification determines that the functor's module provides the features `add`, `get`, `getAll`, and `remove`, where the value of each feature is given by the variable after the following colon. The values of these variables are then computed by the functor's body.

For convenience, the export specification above may also be written more succinctly as follows:

10c ⟨**Export specification for DB.oz (with syntactic sugar)** 10c⟩≡
```
export
   Add
   Get
   GetAll
   Remove
```

The shortcut to just use a variable identifier starting with a capital letter, defines both the variable identifier as well as the feature. The feature is the variable identifier with its first character changed to lowercase.

The functor body is of less importance to us here, however, you can find it in Section A.1. One advantage of modular program development is that during the design of an application one may concentrate first on finding the right interfaces, and only then provide corresponding implementations.

Even though the functor does not import any module, it uses predefined procedures (for example, `Dictionary.new` to create a new dictionary). The compiler provides a set of variable identifiers, that refer to the basic operations on all primitive Oz data types. This set of identifiers is known as the base environment and is documented in detail in *"The Oz Base Environment"*.

When a functor definition is compiled, all free variable identifiers must be bound by the base environment.

2.4 The Graphical Input Form

The functor that implements the graphical form to book flights has the following structure, and its definition resides in file `Form.oz`:

11a ⟨**Form.oz** 11a⟩≡

```
functor
import
   Tk
export
   Book
define
   proc {Book Fs ?Get}
      %% Takes a list of flights and returns the booked flight
      %% and information on the booking user
      ⟨Implementation of Book 80a⟩
   end
end
```

2.5 The Root Functor

The root functor for our last minute flights application uses the previously defined functors that maintain the data base and that provide the user form. The root functor's definition resides in file `LMF.oz`:

11b ⟨**LMF.oz** 11b⟩≡

```
functor
import
   DB Form            % User defined
   System Application % System
define
   %% Enter some flights
   {ForAll ⟨Sample flights 81a⟩ DB.add}
   %% Book until all flights sold out
   proc {Book}
      case {DB.getAll}
      of nil then
         {System.showInfo 'All flights sold.'}
      [] Fs then
         O={Form.book Fs}
      in
         {System.showInfo ('Booked: '#O.key#
                           ' for: '#O.first#
                           ' '#O.last#
                           ' ('#O.email#')')}
         {DB.remove O.key}
         {Book}
      end
```

```
   end
   {Book}
   {Application.exit 0}
end
```

2.6 Compilation

Functors also are compilation units. Each functor definition is compiled separately. For our example, the following sequence of commands

```
ozc -c DB.oz -o DB.ozf
ozc -c Form.oz -o Form.ozf
ozc -c LMF.oz -o LMF.oza
```

compiles our example functor definitions. If you now change the functor definition in, say, `DB.oz` but the interface of the created functor remains the same, none of the other functor definitions need recompilation.

Note that we have chosen as file extensions for pickled functors that are not supposed to be run as applications the string `ozf`. For the root functor of our application we chose `oza`. This is completely transparent as it comes to the semantics of our program, it is just a convention that makes it easier to tell apart which pickled functors are root functors of applications.

2.7 Execution

As before, we just execute the root functor of our application by applying the `ozengine` command to `LMF.oza`:

```
ozengine LMF.oza
```

The next chapter (Chapter 5) explains how applications that consist of several functors are executed.

3

Module Managers

So far we only dicussed how functor definitions can be developed in a modular fashion. This chapter explains how modular applications are executed.

The chapter is kept informal, a more detailed explanation of module managers and how module managers link applications can be found in [1].

3.1 Overview

A module manager maintains a module table. The module table maps urls (or to be more precise, full module urls) to modules or futures to modules.

A module manager links a module at url *U* as follows:

1. If *U* is already in the module table, the module manager returns the entry in the module table.
2. If *U* is not yet in the module table, the module manager creates a new future *M* and stores *M* under key *U* in the module table. As soon as the value of future *M* gets requested, the module manager installs a module from the url *U*.

Linking is done lazily: only when the value of the module is requested (usually implicitly, when the application attempts to access an exported feature), the module gets installed. A module with full module url *U* is installed as follows, where the installation procedure returns a module *M*:

1. The pickled functor stored at *U* is loaded.
2. The module manager computes the full module urls for its imports. This step is detailed in Section 3.3 and Section 3.4.
3. It recursively links all functors with the full module urls computed before. This yields the argument modules.
4. It applies the functor body to the argument modules, which returns the module *M*.

3.2 Lazy Linking

In the above section we have discussed that modules are subject to lazy linking: Only when a module is requested, the functor to compute that module gets loaded.

Lazy linking has dramatic effect on the startup time of an application: applications can start small in a fraction of a second and load additional functionality on demand. And all this happens completely transparently.

System modules are also subject to dynamic linking. Even though module managers give the impression that the system modules are always there, they are also loaded on demand. An application that takes good advantage of that fact is for example the Oz Programming Interface which starts quickly even though it imports all system modules.

3.3 Import Specifications

The entire import part of a functor is called the import specification. Each import specification consists of several import clauses. Each import clause consists of a mandatory module name and an optional module url that must be preceded by the `at` keyword.

For example, in the import specification

```
import
   DB
   MyForm at 'Form.ozf'
```

the first import clause is `DB` which just consists of the module name `DB` and does not provide a module url. The second import clause is `MyForm at 'Form.ozf'` which consists of the module name `MyForm` and the module url `'Form.ozf'`.

The first step in linking a functor is that the module manager computes for each import clause the full module url. This is done according to the following rules:

1. If the import clause features a module url *U*, the full module name is *U*.
2. If the import clause consists of a module name *M* only, where *M* is the name of a system module, the full url is `x-oz://system/`*M*, that is the module name prefixed by `x-oz://system/`.
3. If the import clause consists of a module name *M* only and *M* is not the name of system module, the full url is *M*`.ozf`, that is the module name suffixed by `.ozf`.

All currently defined system modules are listed in Figure 3.1.

3.4 Url Resolution

Given the full url names for each import clause, the module manager determines the urls from which functors are to be loaded by url resolution. Url resolution takes two urls (one is called the base url and the other is called the local url) as input and computes a new resolved url. The process of url resolution you already know from organizing

Figure 3.1 System modules.

Name	Description	Documentation
Application Programming		
`Application`	Command processing and application termination	Chapter *Application Support:* `Application`, *(System Modules)*
`Module`	Module managers	Chapter *Module Managers:* `Module`, *(System Modules)*
Constraint Programming		
`Search`	Search engines	Chapter *Search Engines:* `Search`, *(System Modules)*
`FD`	Finite domain propagators and distributors	Chapter *Finite Domain Constraints:* `FD`, *(System Modules)*
`Schedule`	Scheduling propagators and distributors	Chapter *Scheduling Support:* `Schedule`, *(System Modules)*
`FS`	Finite set propagators and distributors	Chapter *Finite Set Constraints:* `FS`, *(System Modules)*
`RecordC`	Feature constraints (record constraints with open arities)	Chapter *Feature Constraints:* `RecordC`, *(System Modules)*
`Combinator`	Deep-guard combinators	Chapter *Deep-guard Concurrent Constraint Combinators:* `Combinator`, *(System Modules)*
`Space`	First-class computation spaces	Chapter *First-class Computation Spaces:* `Space`, *(System Modules)*
Distributed Programming		
`Connection`	Connecting to running Oz processes	Chapter *Connecting Computations:* `Connection`, *(System Modules)*
`Remote`	Remote module managers	Chapter *Spawning Computations Remotely:* `Remote`, *(System Modules)*
`URL`	URL parsing and resolution routines	Chapter *Referring To Distributed Entities:* `URL`, *(System Modules)*
`Resolve`	URL resolving	Chapter *Resolving URLs:* `Resolve`, *(System Modules)*
`Fault`	Handling faults in distributed programs	Chapter *Detecting and Handling Distribution Problems:* `Fault`, *(System Modules)*
Open Programming		
`Open`	Support for files, sockets, and pipes	Chapter *Files, Sockets, and Pipes:* `Open`, *(System Modules)*
`OS`	POSIX compliant operating system support	Chapter *Operating System Support:* `OS`, *(System Modules)*
System Programming		
`Pickle`	Saving and loading of persistant values	Chapter *Persistent Values:* `Pickle`, *(System Modules)*
`Property`	Querying and configuring engine properties	Chapter *Emulator Properties:* `Property`, *(System Modules)*
`Error`	Error handling routines	Chapter *Error Formatting:* `Error`, *(System Modules)*
`Finalize`	Automatic garbage collection for native entities	Chapter *Memory Management:* `Finalize`, *(System Modules)*
`System`	Miscellaneous system related procedures (printing)	Chapter *Miscelleanous System Support:* `System`, *(System Modules)*
Window Programming		
`Tk`	Classes and procedures for window programming	Chapter *The Module* `Tk`, *(System Modules)*
`TkTools`	Predefined abstractions to han-	Chapter *Graphical Tools:* `TkTools`, *(Sys-*

your html-pages: a url for a relative href-link is resolved with respect to the base url of the containing document.

In this respect, a module manager will behave similarly to a web-browser. When it installs a module that was loaded from url *U*, the latter's full import urls are resolved with respect to *U*. Figure 3.2 shows a small example, where the root functor has the absolute filename `/home/schulte/A.ozf`.

Figure 3.2 Example for resolving urls.

Name	Resolved url	Import specification
A	`/home/schulte/A.ozf`	`import B at 'down/B.ozf' C at 'http://www.foo.org/C.ozf'`
B	`/home/schulte/down/B.ozf`	empty
C	`http://www.foo.org/C.ozf`	`import D at 'D.ozf'`
D	`http://www.foo.org/D.ozf`	empty

3.5 User-defined Module Managers

When the engine starts it has the root module manager that executes the root functor and subsequently links imported functors. However, in many cases it is desirable that applications can create private module managers that just link particular functors.

As an example, suppose we want to use more than a single data base as implemented by the functor `DB.ozf` as shown in Section 2.3

In the Oz Programming Interface, we can link `DB.ozf` twice with two new and different module managers as follows:

```
[DBA]={Module.link ['DB.ozf']}
[DBB]={Module.link ['DB.ozf']}
```

Both `DBA` and `DBB` refer to two independent data bases.

You can observe lazy linking easily. Browsing `DBA` as follows

```
{Browse DBA}
```

shows `DBA<Future>`, which means that `DBA` still refers to a future. Requesting the module by, for example, adding an entry to the data base is also reflected in the Browser: the display of `DBA<Future>` is replaced by a representation of the module's value.

That both module managers work independently can be verified by browsing `DBB`.

`Module.link` takes a list of urls, creates a new module manager and maps each url *U* to the module created by linking the functor at *U*. Reference documentation can be found in Chapter *Module Managers:* `Module`, *(System Modules)*.

3.6 Functors in the Oz Programming Environment

Functors are first class entities in the language that can of course also be created in the Oz Programming Interface. This eases development of functors considerably.

Suppose the following demo functor definition

```
declare
functor F
export Pam
define
   fun {DoPam Xs P Ys}
      case Xs of nil then Ys
      [] X|Xr then {DoPam Xr P {P X}|Ys}
      end
   end

   fun {Pam Xs P}
      {DoPam Xs P nil}
   end
end
```

After feeding the definition, the defined functor can be applied as follows:

```
[M]={Module.apply [F]}
```

The module `M` can be used in the OPI as usual, that is

```
{Browse {M.pam [1 2 3] fun {$ I} I+1 end}}
```

displays the list `[4 3 2]`.

`Module.apply` takes a list of functors as argument, creates a new module manager, applies each functor element and returns the resulting list of modules. It is also possible to specify the base url used in linking the argument modules of the applied functor. For more information see Chapter *Module Managers:* `Module`, *(System Modules)*.

Of course, also other situations allow to take advantage of first-class functors and that they can be applied by module managers. In particular they are useful for remote module managers that create new Oz processes on networked computers. You can learn more on this issue in *"Distributed Programming in Mozart - A Tutorial Introduction"*.

4

Pickles for Persistent Data Structures

Applications often require to store their state on file and load the saved data later. Oz supports this by pickling of data structures: Data structures are made persistent by writing them to files.

4.1 Stateless, Stateful, and Sited Nodes

Values, or more precisely nodes,[1] in Oz are either stateless or stateful:

Stateful Basic data structures that are stateful include cells, variables, and ports. Since objects, arrays and dictionaries are conceptually composed of cells, they are stateful as well.

Stateless Stateless data structures are literals, numbers, records, classes, chunks, and procedures.

In addition, nodes in the store can be sited: the node is specific to a particular site; it is a site-bound resource. For example, classes for files (`Open.file`) and widget classes for graphics (for example, `Tk.toplevel`) are sited.

Only stateless and un-sited nodes can be made persistent by pickling.

4.2 Loading and Saving Values

After executing the following statement

```
X=a(proc {$ Y} Y=X end 1 2 f:X)
```

`X` refers to a record node. The node can be saved or pickled to the file `test.ozp` by executing

```
{Pickle.save X 'test.ozp'}
```

Pickling traverses the entire graph reachable from the root node (which is referred to by `X` in our example), creates a portable description of the graph and writes the description to a file.

The pickled data structure can be loaded by

[1] A node may also be an unbound variable, i.e. a value that is not yet determined

```
Z={Pickle.load 'test.ozp'}
```

Now `Z` refers to a graph which is an isomorphic clone of the graph that has been saved. For our example this means: what can be reached from `X` and `Z` is equal. For example

```
X.1==Z.1
```

evaluates to `true`. In fact, `X` and `Z` cannot be distinguished.

Loading of pickles works across the internet: it is possible to give a url rather than just a filename. For example, if you have a public html directory `~/public_html` and you move the pickle file `test.ozp` there, everybody can load the pickle across the internet. Suppose that the url of your public html directory is `http://www.ps.uni-sb.de/~schulte` then the pickle can be loaded by

```
Z={Pickle.load 'http://www.ps.uni-sb.de/~schulte/test.ozp'}
```

4.3 Example: The Data Base Revisited

To extend the data base we developed in Section 2.3 with persistence, we just add two procedures to load and save a data base and extend the export specification accordingly. The toplevel structure of the functor definition is as follows:

20a ⟨**PDB.oz** 20a⟩≡

```
functor
  import Pickle
  ⟨Export specification for PDB.oz 20b⟩
  ⟨Body for PDB.oz 20c⟩
end
```

The functor imports the system module `Pickle`. The export specification is just extended by the fields `load` and `save`.

20b ⟨**Export specification for PDB.oz** 20b⟩≡

```
⟨Export specification for DB.oz 10b⟩
   load: Load
   save: Save
```

The body for `PDB.oz` is as follows:

20c ⟨**Body for PDB.oz** 20c⟩≡

```
⟨Body for DB.oz 79a⟩
   proc {Save File}
      {Pickle.save {Ctr get($)}#
                   {Dictionary.toRecord db Data}
       File}
   end
   proc {Load File}
      I#D={Pickle.load File}
```

```
in
   {Dictionary.removeAll Data}
   {Ctr init(I)}
   {Record.forAllInd D
    proc {$ K E}
       {Dictionary.put Data K E}
    end}
end
```

`Save` takes as input the filename of the pickle, whereas `Load` takes the url from which the pickle can be loaded.

When using the persistent data base, it has to be kept in mind that it does not offer concurrency control: Simultaneous add and remove, as well as load and save operations performed by several threads might leave the data base in an inconsistent state. In ??? we will develop the data base in a data base server that also allows for concurrency control.

Note that since we only extended the functionality the functor provides, all programs that used the non-persistent data base could, in principle, still use the persistent data base with out being recompiled. We say *could* because the implementaion of the persistent database is named `PDB.ozf` rather than `DB.ozf`. However, you can give it a try and simply rename `PDB.ozf` to `DB.ozf`: all applications based on the the non-persistent implementation will continue to work as before but now using the persistent implementation (though without actually taking advantage of the persistency).

4.4 Pickle Compression

Pickles can also be compressed so that they occupy less space on disk. For example, a compressed pickle for `X` can be written to file `testz.ozp` by

```
{Pickle.saveCompressed X 'testz.ozp' LevelI}
```

`LevelI` is an integer between `0` and `9` specifying the compression level: the higher the value the better the compression, but the longer pickling takes. A value of `0` gives no compression.

Compression time and ratio depend on the data being pickled. The compression ratio might vary between 20 and 80 percent, while compression at level 9 is usually less than 2 times slower than using no compression.

5

More On Functors

5.1 Computed Functors

We distinguish between compiled functors and computed functors. A compiled functor is obtained by compilation of a functor definition. Computed functors are obtained by executing compiled functors whose definitions contain nested functor definitions. Compiled functors can only have lexical bindings to the data structures of the base environment. Computed functors can have lexical bindings to all data structures that the creating compiled functors supply to their definitions.

Pickled computed functors can carry computed data structures with them. This matters since

1. a computed data structure can now be loaded together with a functor rather than being computed a new for each virtual machine using it.
2. the functors needed to compute the carried with data structure are not needed by the virtual machine using it.

Computed functors are syntactically supported by a **prepare** and **require** section. For example, the root functor definition in the file `LMF.oz` can be rewritten using a **prepare** section as follows:

```
functor
import
   DB Form            % User defined
   System Application % System
prepare
   Flights = ⟨Sample flights 81a⟩
define
   %% Enter some flights
   {ForAll Flights DB.add}
   ...
end
```

Here the difference between the compiled functor and the computed functor is that the compiled functor contains the code to create the list of sample flights. The computed functor just contains the list itself.

All variable identifiers that are introduced in the `prepare` section are visible in the `define` section. The variables introduced by the `import` section are of course only visible in the `define` section.

The `require` section of a computed functors relates to the `prepare` section as does the `import` section to the `define` section: modules imported in the `require` section are available in the `prepare` section.

5.2 Imports

`import` and `require` specifications support features and fields. For example, in the main functor for our last minute flight booking system, we could have written the import clause for `DB` as follows:

```
DB(add getAll remove)
```

Besides of the documentational advantage of explicitly listing the features, the compiler tries to enforce that only listed features are used for dot access. For example, given the above import clause, the following access

```
DB.addd
```

raises an error during compilation.

In addition, also variables can be given as fields in the import specification as follows:

```
DB(add:Add getAll:GetAll remove)
```

The variables introduced for the fields interact with dynamic linking as follows: The module is requested as soon as the value for one of the variables is requested.

6

Application Deployment

6.1 Linking Functors

Application development can be considerably eased by splitting the application in a large number of orthogonal and reusable functors. However, deployment of an application gets harder in the presence of a large number of functors:

1. Installing the application requires correct installation of a large number of functors.
2. Execution might be slow due to frequent file- or even network accesses.

The commandline tool `ozl` eases deployment by creating a new functor that includes imported functors in a prelinked fashion: it is possible to collapse a collection of functors into a single equivalent one. The model that should be kept in mind, is that the newly created functor employs an internal, private module manager that executes the toplevel application functor together with all included functors.

The linker can be invoked on the input functor `In` in order to create an output functor `Out` as follows:

```
ozl In -o Out
```

For example, from the pickled toplevel functor `LMF.ozf` a new functor can be created as follows:

```
ozl LMF.ozf -o LMF.oza
```

where the pickled functor `LMF.ozf` is created by compilation as follows:

```
ozc -c LMF.oz -o LMF.ozf
```

Now the newly created pickled functor `LMF.oza` can be installed everywhere, the functors stored in `DB.ozf` and `Form.ozf` are included in `LMF.oza`.

The linker can be used in verbose mode with the option `-verbose` (or `-v` as abbreviation). In verbose mode the linker prints information on which functors are included and which functors are imported by the newly created functor. For example,

```
ozl -v LMF.ozf -o LMF.oza
```

prints something like

```
Include:
   /home/schulte/DB.ozf, /home/schulte/Form.ozf,
   /home/schulte/LMF.ozf.
Import:
   x-oz://system/Application.ozf, x-oz://system/System.ozf,
   x-oz://system/Tk.ozf.
```

The linker also supports options that control which functors are included, for more information see Chapter *The Oz Linker:* `ozl`, *(Oz Shell Utilities)*.

6.2 Compressed Pickled Functors

Pickles created by the compiler and linker can also take advantage of compression. For that matter, both tools support the `-compress` (or `-z` as shortcut) option that must be followed by a single digit that defines the compression level to be used.

For example, the pickled functor `LMF.oza` can be created compressed by

```
ozl --compress 9 LMF.ozf -o LMF.oza
```

This reduces the used disk space by 50%.

6.3 Executing Functors

This section shows a convenient form to execute functors.

The option `-exec` (or `-x` as shortcut) can be supplied to both compiler and linker. Functors that are created with that option can be directly executed. For example, the file `lmf.exe` created with

```
ozl -x LMF.ozf -o lmf.exe
```

can be directly executed:

```
lmf.exe
```

The pickled functor `lmf.exe` just features a particular header that allows direct execution. It can still be used together with the `ozengine` program:

```
ozengine lmf.exe
```

Naturally, the extension `.exe` can be omitted under Unix.

Part II

Programming With Concurrency

7

Concurrency For Free

This part of the tutorial addresses the following theme: what happens to programming when support for concurrency is extremely cheap, economical, and efficient. Suddenly, an entirely different style of programming and design is made possible. We are going to explore and exploit this new freedom.

Oz has very efficient, very economical, very lightweight threads, with fair preemptive scheduling. We don't mean that Oz threads are just somewhat better than brand X; we mean that brand X can't even see our dust with a telescope, er... well, just about anyway! In order to assuage the skeptics, we first exhibit a program that demonstrates massive concurrency and exercises the worst case. Doubters are encouraged to throw that program at their favorite programming language... and watch it die, eventually. Meanwhile, you could mount a clay tablet device, and engage in the more rewarding exercise of installing Windows from sumerian backup.

The program is invoked with:

```
death --threads N --times M
```

and creates *N* threads. Each thread does nothing but *yield* immediately. Normally we would let the preemptive scheduler take care of interrupting a thread to switch to a new one, but here, in order to exercise the worst case, as soon as a thread is allowed to run, it explicitly *yields*. Thus the program does little else but switch between threads. Each thread yields *M* times and then terminates. When all threads have terminated, the program also terminates.

I just tried the following:

```
death --threads 10000 --times 10
```

In other words, 10000 threads are created and must each yield 10 times. This results in 100000 thread switches. It takes 3s on this little steam-driven laptop. I have run the same program on a real computer at the lab but using:

```
death --threads 100000 --times 10
```

It takes 7.5s. There are 100000 threads at all time runnable, and they must perform 1000000 thread switches. Try creating 100000 threads in Java... really, go ahead, I insist! I promise not to laugh!

Just so you don't have to take my word for it, I coded the same program in Java and tried:

```
java Death 1000 10
```

This takes 2:40mn!

What was the point of this exercise? It was not prove that Oz is better than Java; in this respect the test above was deliberately unfair: Java was never intended to support designs with massive concurrency... and *that* is the point. Oz was from the start designed as a platform for concurrent computation. That concurrency is so cheap and efficient makes entirely new designs possible that would not be realistic or even conceivable in other languages. Whenever you need to perform an operation asynchronously you simply spawn a new thread. You can design your application as a collection of concurrent objects or agents, etc...

Death by Concurrency in Oz

Here is the code of the Oz application used in the benchmark above:

```
functor
import Application
define
   proc {Yield} {Thread.preempt {Thread.this}} end
   proc {Run N}
      {Yield}
      if N>1 then {Run N-1} end
   end
   Args = {Application.getCmdArgs
           record(threads(single type:int optional:false)
                  times(  single type:int optional:false))}
   proc {Main N AllDone}
      if N==0 then AllDone=unit else RestDone in
         thread {Run Args.times} AllDone=RestDone end
         {Main N-1 RestDone}
      end
   end
   {Wait {Main Args.threads}}
   {Application.exit 0}
end
```

Death by Concurrency in Java

Here is a very similar program, in Java:

```
import java.lang.*;
class MiniThread extends Thread {
    int n;
    MiniThread(int m) { n=m; }
    public void run() {
        do { yield(); n--; } while (n>0);
```

```
    }
}
public class Death {
    public static void main(String[] argv) {
        int threads = Integer.parseInt(argv[0]);
        int times   = Integer.parseInt(argv[1]);
        for(int i=threads;i>0;i--) {
            MiniThread t = new MiniThread(i);
            t.start();
        }
    }
}
```

8

Programming Patterns

In this chapter, we present a number of patterns that take advantage of concurrency. When programming in Oz, you don't have to agonize over the question whether you really need to invest into a new thread. You just do it! This bears repeating because most people with experience of threads in other languages just don't believe it. Threads aren't just for long running computations: you can spawn threads to perform single operations asynchronously.

In the previous chapter, we demonstrated that it is realistic to create a huge number of threads. However, we exercised the worst case: all threads wanted a piece of the action all the time. In reality, the situation is usually much better: most threads are blocked, waiting for some event, and only a very small number of them compete for processor time.

8.1 Stream Processing Agents

A very common pattern is for a thread to implement an agent that processes all messages that appear on a stream. For example, here, procedure `Process` is applied to each element of stream `Messages`, one after the other:

```
thread {ForAll Messages Process} end
```

Typically, the tail of the stream is uninstantiated, at which point the `ForAll` procedure, and thus the thread, suspends until a new message comes in that instantiates the stream further.

8.1.1 Stream Merging

As an application of this technique, we consider now the *fair merge* of two streams `L1` and `L2` into one single new stream `L3`. For this, we create the new port `Mailbox` connected to stream `L3`, and two agents to forward the messages of `L1` and `L2` to the `Mailbox`:

```
proc {Merge L1 L2 L3}
   Mailbox = {Port.new L3}
   proc {Forward Msg} {Port.send Mailbox Msg} end
in
```

```
      thread {ForAll L1 Forward} end
      thread {ForAll L2 Forward} end
   end
```

Fairness of merging is guaranteed by the fairness of thread scheduling. Actually, the code above can easily be generalized. Here is an abstraction that returns two results: a merged stream `L` and a procedure `AlsoMerge` to cause yet another stream to be merged into `L`:

```
proc {MakeMerger AlsoMerge L}
   Mailbox = {Port.new L}
   proc {Forward Msg} {Port.send Mailbox Msg} end
in
   proc {AlsoMerge LL}
      thread {ForAll LL Forward} end
   end
end
```

8.2 Communication Patterns

A great advantage of *concurrency for free* is that it gives you a new way to manage design complexity: you can partition your design into a number of small simple agents. You then use streams to connect them together: agents exchange and process messages.

There are two major designs for stream-based communication among agents: one is the *email* model, the other the *newsgroup* model. Of course, in realistic applications, you should mix these models as appropriate.

8.2.1 Email Model

In the email model, each agent is equipped with his own mailbox. In the simplest case, the agent is known to others only through its mailbox. For example, here is a function that takes a message processing function as argument, creates an agent, and returns its mailbox:

```
proc {MakeAgent Process Mailbox}
   thread {ForAll {Port.new $ Mailbox} Process} end
end
```

8.2.2 Newsgroup Model

In the newsgroup model, all agents process and post to the same stream of messages.

8.2.2.1 Forward Inference Engine: Implementation

We illustrate the newsgroup model with an application to forward inference rules. A forward inference rule has the form $\forall \bar{x}\, C \Rightarrow D$ where C and D are conjunctions of literals and all variables of the conclusion appear in the premise. The newsgroup will

be where inferred literals are published. A rule is said to be partially recognized when some, but not all, of the premise literals have been discovered on the newsgroup. A partially recognized rule is implemented by an agent that reads the newsgroup in search of candidates for the next premise literal. When a rule has been fully recognized, its conclusion is then *asserted*, which normally results in the publication of new literals.

We express the engine in the form of a functor that exports the list of `Literals` being published as well as a procedure to `Assert` literals and rules.

35a ⟨**Forward Inference Module** 35a⟩≡

```
functor
import Search
export Literals Assert
define
   Literals Box={Port.new Literals}
   ⟨Assert conclusion 35b⟩
   ⟨Replace symbolic by actual variables 36b⟩
   ⟨Agent for partially recognized rule 36c⟩
end
```

What can be asserted are literals and rules, and conjunctions thereof. A conjunction is represented as a list. Since we don't want to publish twice the same literal (or else we might have termination problems), we maintain here a database of all published literals, indexed according to their outermost predicate. A real implementation might prefer to replace this by an adaptive discrimination tree. Whenever we are about to publish a literal, we first check that it isn't already in the database: in that case, we enter it and then only publish it.

35b ⟨**Assert conclusion** 35b⟩≡

```
Database = {Dictionary.new}
proc {Assert Conclusion}
   if {IsList Conclusion} then {ForAll Conclusion Assert}
   elsecase Conclusion of rule(VarList Premises Conclusion) then
      ⟨Assert rule 36a⟩
   else
      Pred = {Label Conclusion}
      Lits = {Dictionary.condGet Database Pred nil}
   in
      if {Member Conclusion Lits} then skip else
         {Dictionary.put Database Pred Conclusion|Lits}
         {Port.send Box Conclusion}
      end
   end
end
```

Asserting a rule consists of creating an agent to recognize it. The agent is equipped with (1) an index (2) a predicate. The index indicates which premise literal to recognize next; it starts from `N`, the last one, and decreases down to 1. The predicate constrains a representation `rule(premises:P conclusion:C)` of the partially recognized rule. In order to create this representation, we invoke `Abstract` to replace the quantified symbolic variables of the rule by new free Oz variables.

36a ⟨**Assert rule** 36a⟩≡

```
local
   N = {Length Premises}
   proc {RulePredicate RuleExpression}
      case {Abstract VarList Premises#Conclusion} of P#C then
         RuleExpression=rule(premises:P conclusion:C)
      end
   end
in {Agent N RulePredicate} end
```

Below, we create a mapping from symbolic variables to new free Oz variables, then recursively process the expression to effect the replacements. Note that we cary the list `Avoid` of symbolic variables that are quantified in a nested rule expression.

36b ⟨**Replace symbolic by actual variables** 36b⟩≡

```
fun {Abstract VarList E}
   Vars = {Record.make o VarList}
   fun {Loop E Avoid}
      if {IsAtom E} then
         if {Member E Avoid} then E
         elseif {HasFeature Vars E} then Vars.E
         else E end
      elseif {IsRecord E} then
         case E of rule(VarL Prem Conc) then
            rule(VarL {Loop Prem {Append VarL Avoid}}
                      {Loop Conc {Append VarL Avoid}})
         else {Record.map E fun {$ F} {Loop F Avoid} end} end
      else E end
   end
in {Loop E nil} end
```

The agent is equipped with the index `I` of the next premise literal to be recognized and with `RulePredicate` to constrain the representation of the partially recognized rule. For each literal that is being published, the agent finds all possible solutions that result from unifying it with the `I`th premise literal, and produces the corresponding refined predicates. For each new predicate produced, a new agent is created to recognize the next premise literal; unless of course all premise literals have been recognized, in which case we retrieve the corresponding instantiated conclusion and assert it.

36c ⟨**Agent for partially recognized rule** 36c⟩≡

```
proc {Agent I RulePredicate}
   {ForAll Literals
    proc {$ Literal}
       {ForAll
        {Search.allP
         proc {$ RuleExpression}
            {RulePredicate RuleExpression}
            {Nth RuleExpression.premises I}=Literal
         end 1 _}
        proc {$ NewRulePredicate}
```

```
                  if I==1 then {Assert {NewRulePredicate}.conclusion}
                  else thread {Agent I-1 NewRulePredicate} end end
               end}
            end}
         end
```

8.2.2.2 Forward Inference Engine: Usage

The functor can be compiled as follows:

```
ozc -c forward.oz
```

and you might experiment with it in the OPI as follows (where *DIR* is the directory where the compiled functor is located).

```
declare [Forward] = {Module.link ['DIR/forward.ozf']}
{Browse Forward.literals}
{Forward.assert rule([x y z] [a(x y) a(y z)] a(x z))}
{Forward.assert a(one two)}
{Forward.assert a(two three)}
```

We asserted one rule expressing the transitivity of binary predicate `a`, and then two facts. In the browser, you will now observe:

```
a(one two)|a(two three)|a(one three)|_<Future>
```

8.3 Synchronization

In Oz, synchronization is done on data and typically takes the form of waiting for a variable to become instantiated. Furthermore, this happens automatically: every operation that requires *determined* data will suspend until this data becomes determined. For example, this is why you can write:

```
{ForAll Messages Process}
```

where `Messages` is a stream whose tail only incrementally becomes instantiated with new messages. The `ForAll` operation suspends when it reaches the uninstantiated tail of the stream, and resumes automatically when further messages become available.

If you need to synchronize explicitly on a variable `X`, you may write:

```
{Wait X}
```

which suspends this thread until `X` becomes determined.

The truth is actually much more general: a conditional suspends until its condition can be decided, one way or the other. What makes this possible is the fact that the information in the *constraint store* increases monotonically. A conditional suspends until its condition is *entailed* by the store (implied), or *disentailed* (its negation is implied). Thus, the `Wait` operation mentioned above can (almost) be coded as follows:

```
proc {Wait X}
   if X==a then skip else skip end
end
```

This suspends until it can be decided whether or not `X` is equal to `a`. I said 'almost' because in between being free and determined, a variable may be kinded (i.e. its type is known), and the code above does not account for this possibility.

The `ForAll` procedure is actually implemented as follows:

```
proc {ForAll L P}
   case L of H|T then {P X} {ForAll T P}
   elseof nil then skip end
end
```

The case statement (a conditional) suspends until it can be determined whether `L` matches `H|T`, i.e. is a list pair.

8.3.1 The Short-Circuit Technique

The short-circuit technique is the standard means of programming an n-way rendez-vous in concurrent constraint programming. The problem is the following: given n concurrent threads, how to synchronize on the fact that they have all terminated? The idea is to have a *determined* termination token, and to require that each thread, when it terminates, passes the token that it got from its left neighbour to its right neightbour. When the termination token really arrives at the *rightmost* end, we know that all threads have terminated.

For example, in the example below, we create `Token0` with value `unit`, and then each thread, when it terminates, passes the token on to the next thread. When the value `unit` reaches `Token5`, we know that all threads have terminated.

```
local Token0 Token1 Token2 Token3 Token4 Token5 in
    Token0 = unit
    thread ... Token1=Token0 end
    thread ... Token2=Token1 end
    thread ... Token3=Token2 end
    thread ... Token4=Token3 end
    thread ... Token5=Token4 end
in
    %% synchronize on the termination of all 5 threads
    {Wait Token5}
end
```

This technique was used in Section 7. Of course it can be used for any arbitrary n-way rendez-vous, and not exclusively for synchronizing on the termination of a collection of threads.

Part III

Client/Server Applications

9

Introduction

A large fraction of client/server applications fall in the same simple pattern: there is a basic service encapsulated as an object and we wish to allow remote clients to send requests to this object, to be processed at the server host.

The basic idea is to make available to clients a procedure that forwards a client's request to the live server object. This forwarding is effected by means of a port. The forwarding procedure itself is made available indirectly through a ticket. This ticket is placed in a file that is accessible through a URL.

9.1 A Generic Server `server.oz`

It is straightforward to write a generic server module that exports a `Start` procedure. The latter takes 2 arguments: `Proc` the object or procedure implementing the service and `File` the name of the file where the ticket should be saved. `Proc` is intended to be applied to messages forwarded by clients.

The forwarding procedure `Proxy` takes the clients message `Msg` and sends `request(Msg OK)` to the server's port. The server binds `OK` to true or false depending on whether the `Msg` is processed successfully or an exception is raised.

```
functor
import Connection Pickle
export Start
define
   proc {Start Proc File}
      Requests P = {NewPort Requests} Ticket
      proc {Proxy Msg}
         if {Port.send P request(Msg $)} then skip
         else raise remoteError end end
      end
   in
      {New Connection.gate init(Proxy Ticket) _}
      {Pickle.save Ticket File}
      {ForAll Requests
       proc {$ R}
          case R of request(Msg OK) then
             try {Proc Msg} OK=true catch _ then
```

```
                  try OK=false catch _ then skip end
               end
            else skip end
         end}
      end
   end
```

The server functor will be used as an import in subsequent examples and can be compiled as follows:

```
ozc -c server.oz
```

10

Registry Application

An example application is where the service is a shared registry. A client can connect to the registry server and add or lookup an entry. The registry is simply a dictionary.

10.1 Server `db-server.oz`

The registry server is compiled as follows:

```
ozc -x db-server.oz -o db-server.exe
```

and can be started with the command line:

```
db-server.exe --ticketfile file
```

Initially, it has an empty registry.

```
functor
import
   Server at 'server.ozf'
   Application
define
   class Registry
      feat db
      meth init {Dictionary.new self.db} end
      meth put(Key Val) {Dictionary.put self.db Key Val} end
      meth get(Key Val) {Dictionary.get self.db Key Val} end
      meth condGet(Key Default Val)
         {Dictionary.condGet self.db Key Default Val}
      end
   end
   DB = {New Registry init}
   Args = {Application.getCmdArgs
           record(
              ticketfile(single char:&t type:string optional:false))}
   {Server.start DB Args.ticketfile}
end
```

10.2 Client `db-client.oz`

The client loads the pickled ticket from the given URL and uses it to obtain from the server the forwarding procedure. The client can be compiled as follows:

```
ozc -x db-client.oz -o db-client.exe
```

and can be invoked in one of two ways:

```
db-client.exe --url=URL --get=KEY
db-client.exe --url=URL --put=KEY VAL
```

The first form retrieves a entry from the registry and displays it on standard output. The second form stores an entry in the registry.

```
functor
import
   Application Connection System Pickle
define
   Args = {Application.getCmdArgs
           record(
              url(single type:string optional:false)
              get(single type:atom)
              put(single type:atom))}
   DB   = {Connection.take {Pickle.load Args.url}}
   if     {HasFeature Args get} then
      {System.showInfo {DB get(Args.get $)}}
   elseif {HasFeature Args put} then
      case Args.1 of [Value] then
         {DB put(Args.put Value)}
      else
         {System.showError 'Missing value argument'}
         {Application.exit 1}
      end
   else
      {System.showError 'One of --get or --put is required'}
      {Application.exit 1}
   end
   {Application.exit 0}
end
```

11

Compile Server Application

We now develop an application where a client can send an Oz file containing a functor expression to a compile server and gets back the corresponding compiled functor. The server provides a compilation service.

11.1 Server `ozc-server.oz`

The compile server is compiled as follows:

```
ozc -x ozc-server.oz -o ozc-server.exe
```

and can be started with the command line:

```
ozc-server.exe --ticketfile file
```

the server returns `yes(F)` where `F` is a functor value, or `no(Msgs)`, if compilation failed, where `Msgs` are the error messages obtained from the compiler's interface.

```
functor
import
   Compiler Application
   Server at 'server.ozf'
define
   class OZC
      prop locking
      feat engine interface
      meth init
         self.engine    = {New Compiler.engine    init}
         self.interface = {New Compiler.interface init(self.engine)}
         {self.engine enqueue(setSwitch(expression true))}
      end
      meth compile(VS $)
         lock F in
            {self.engine
             enqueue(feedVirtualString(VS return(result:F)))}
            {Wait {self.engine enqueue(ping($))}}
```

```
            if {self.interface hasErrors($)} then
               no({self.interface getMessages})
            else yes(F) end
         end
      end
   end
   Service = {New OZC init}
   Args = {Application.getCmdArgs
           record(
              ticketfile(single char:&t type:string optional:false))}
   {Server.start Service Args.ticketfile}
end
```

11.2 Client `ozc-client.oz`

The client can be compiled as follows:

```
ozc -x ozc-client.oz -o ozc-client.exe
```

and can be invoked with:

```
ozc-client.exe --url=URL --in=InFile --out=OutFile
```

It loads the compile server's ticket from *URL*, uses it to obtain the forwarding procedure, applies it to the textual contents of *InFile* and saves the returned functor value in *OutFile*. Note that we convert the string (i.e. list) representation of the file's contents to a byte string for more efficient transmission; this is not necessary, but greatly reduces the amount of data that needs to be transmitted.

```
functor
import Application Open Pickle Connection
define
   Args = {Application.getCmdArgs
           record(
              url( single type:string optional:false)
              'in'(single type:string optional:false)
              out( single type:string optional:false))}
   File = {New Open.file init(name:Args.'in')}
   Text = {File read(list:$ size:all)}
   {File close}
   OZC  = {Connection.take {Pickle.load Args.url}}
   case {OZC compile({ByteString.make Text} $)}
   of yes(F) then
      {Pickle.save F Args.out}
      {Application.exit 0}
   elseof no(Msgs) then raise ozc(Msgs) end end
end
```

Part IV

Programming the Web

12

Applets

Oz applications can be executed by clicking links on web pages.

12.1 Enabling Oz Applets

In order to start Oz applications by clicking links on web pages, the web browser must be Mozart enabled, which is described in Appendix *Enabling Oz Applets, (Installation Manual)*.

13

Servlets

A *servlet* is a small application that exists on a Web server and that can be invoked by a CGI command. A servlet is usually called a *CGI script*. CGI (Common Gateway Interface) is a protocol that defines how data is passed between a Web server and a servlet.

A servlet is a program that accepts an input and calculates a result. To be precise, it does the following steps:

- Get the arguments for the servlet by calling `Application.getCgiArgs`. A standard application would call `Application.getCmdArgs` for this purpose. The former is used in exactly the same way as the latter, but instead of parsing command line arguments, it parses CGI arguments.
- Calculate the result.
- Print a header on `stdout` that defines the content type. The content type tells the Web browser what the type of the result is, so that it knows how to display it. For example, if the servlet outputs HTML, you have to print something like:

  ```
  'Content-type: text/html\n\n'
  ```

  (without the quotes). The `Open.html` class has support for this (see example below).
- Output the real data. For example, text in HTML format.

The following example follows this structure. It uses a class `Open.html` to generate HTML code on the fly. You can test it by sending a URL that looks like this:

```
'http://www.you.edu/~you/small.cgi?number=10&text=Hi'
```

In this example, the value `10` is passed for the argument `'number'` and the value `"Hi+Guys"` for the argument `'text'` (in CGI argument syntax, the plus is used to quote a space).

```
functor
import Application Open
define
   %% Parse the arguments
```

```
   Args={Application.getCgiArgs
      record(number(single type:int default:0)
             text(single type:string default:"none"))}

   %% A file that supports HTML output
   Out={New class $
               from Open.file Open.html
            end
        init(name:stdout)}

   %% Print MIME content header
   {Out header}

   %% Print HTML output
   {Out tag(html(head(title('My First CGI'))
                 body(bgcolor:'#f0f0e0'
                      h1('My First CGI')
                      p('The number is: '#Args.number)
                      p('The text is: '#Args.text))))}

   %% Terminate
   {Application.exit 0}
end
```

In order to compile this servlet, you have to do:

```
ozc --execpath=OZHOME/bin/ozengine -x small.oz -o small.cgi
```

Where *OZHOME* denotes the installation directory of the Mozart system. The `execpath` argument is needed because the servlet needs an absolute path. Servlets are normally executed by the Web server in an extremely minimal user environment. The user is typically called `nouser` or `www` and has almost no rights. In particular you cannot expect the Mozart system to be in the path of the user! This is why you need an absolute pathname when compiling the servlet.

On a Unix system, you can more simply invoke:

```
ozc --execpath=`which ozengine` -x small.oz -o small.cgi
```

The final step is to install the servlet in your Web server. For this you should contact your local Web site administrator.

Part V

Distributed Applications

14

Chat Application

A chat system permits participants on arbitrary machines on the internet to engage in a real-time text-based discussion. New individual can join or leave the chat forum at any time. This scenario is intended to be realistic, which means that the chat system must be reasonably robust in the face of network failures, as well as machine and process crashes.

In this tutorial application, we will not set out to solve all problems that may be associated with distributed applications; rather, we will demonstrate how simple it is to realize a fully distributed application with reasonable robustness properties.

14.1 Chat Server

The server creates a port `NewsPort` and makes it available through a ticket. The ticket, as usual, is saved into a file which clients normally will load through a url. When a client wants to participate in the discussion forum, it needs not only `NewsPort` in order to post messages, but also the stream of messages that results from all posts to `NewsPort`, in order to display these messages to the user. The server could hand out the stream of all messages from the creation of `NewsPort`, but it seems more desirable to only hand out a stream that has only the messages posted after the client's request to connect to the discussion.

When a client wants to connect to the chat forum, it obtains `NewsPort` by means of the ticket that the server made available at some url, and it posts a message of the form `connect(Messages)`, where `Messages` is a new variable. The server then binds the variable to the stream of messages following the `connect(...)` message.

55a ⟨Chat Server 55a⟩≡

```
functor
import
   Application(getCmdArgs) Connection(gate) Pickle(save)
define
   Args = {Application.getCmdArgs
           record(ticketfile(single type:string optional:false))}
   NewsPort
   local Ticket in
      {New Connection.gate init(NewsPort Ticket) _}
      {Pickle.save Ticket Args.ticketfile}
```

```
      end
      {List.forAllTail {Port.new $ NewsPort}
       proc {$ H|T}
          case H of connect(Messages) then Messages=T else skip end
       end}
   end
```

The server (source in `chat-server.oz`[1]) can be compiled as follows:

```
ozc -x chat-server.oz
```

and invoked as follows:

```
chat-server --ticketfile FILE
```

14.2 Chat Client

The client consists of 2 agents: (1) a user interface agent and (2) a message stream processor.

56a ⟨Chat Client 56a⟩≡

```
functor
import
   Application(getCmdArgs) Pickle(load) Connection(take)
   Viewer(chatWindow) at 'chat-gui.ozf'
define
   Args = {Application.getCmdArgs
           record(url(single type:string optional:false)
                  name(single type:string optional:false)
                 )}
   NewsPort={Connection.take {Pickle.load Args.url}}
   SelfPort
   ⟨Chat Client: obtain and process message stream 56b⟩
   ⟨Chat Client: create user interface agent 57a⟩
   ⟨Chat Client: process message stream 57b⟩
end
```

The client obtains the stream of messages from the server by sending a `connect(...)` message. It then forwards every message on that stream to its internal `SelfPort`. The user interface will also direct messages to this internal port.

56b ⟨Chat Client: obtain and process message stream 56b⟩≡

```
thread
   {ForAll {Port.send NewsPort connect($)}
    proc {$ Msg} {Port.send SelfPort Msg} end}
end
```

[1] `chat-server.oz`

When creating the user interface, we supply it with the internal `SelfPort` so that it may also post internal messages. In this simplistic implementation, the user interface simply posts messages of the form `say(`*String*`)` to request that this *String* be posted to the global chat message stream.

57a ⟨**Chat Client: create user interface agent** 57a⟩≡

```
Chat = {New Viewer.chatWindow init(SelfPort)}
```

Finally, here is where we process all messages on the internal stream. A `msg(FROM TEXT)` message is formatted and shown in the chat window. A `say(TEXT)` message is transformed into `msg(NAME TEXT)`, where `NAME` identifies the user, and posted to the global chat stream; actually `TEXT` is additionally converted into the more compact byte string representation for more efficient transmission.

57b ⟨**Chat Client: process message stream** 57b⟩≡

```
NAME = Args.name
{ForAll {Port.new $ SelfPort}
 proc {$ Msg}
    case Msg of msg(FROM TEXT) then
       {Chat show(FROM#':\t'#TEXT)}
    elseof say(TEXT) then
       {Port.send NewsPort msg(NAME {ByteString.make TEXT})}
    else skip end
 end}
```

The client (source in `chat-client.oz`[2]) can be compiled as follows:

```
ozc -x chat-client.oz
```

and invoked as follows:

```
chat-client --name USER --url URL
```

14.3 Graphical User Interface

The user interface is always what requires the most code. We won't go through the details here (but see the Window Programming Tutorial for extensive information), but merely point out that the `@entry` widget is asked to respond to a `Return` keypress, by invoking the `post` method. The latter posts a `say(`*Text*`)` message to the internal port, where *Text* is the text of the entry as typed by the user. This text is then deleted and the entry can be reused to compose and submit another message.

57c ⟨**Chat GUI** 57c⟩≡

```
functor
import
   Tk Application(exit:Exit)
export
```

[2] `chat-client.oz`

```
   ChatWindow
define
   class ChatWindow from Tk.toplevel
      attr canvas y:0 vscroll hscroll tag:0 selfPort entry quit
      meth init(SelfPort)
         Tk.toplevel,tkInit
         selfPort := SelfPort
         canvas := {New Tk.canvas
                    tkInit(parent:self bg:ivory width:400 height:300)}
         vscroll := {New Tk.scrollbar tkInit(parent:self orient:v)}
         hscroll := {New Tk.scrollbar tkInit(parent:self orient:h)}
         entry   := {New Tk.entry     tkInit(parent:self)}
         quit    := {New Tk.button    tkInit(parent:self text:'Quit'
                                             action:proc{$} {Exit 0} end)}
         {Tk.addYScrollbar @canvas @vscroll}
         {Tk.addXScrollbar @canvas @hscroll}
         {@canvas tk(configure scrollregion:q(0 0 200 0))}
         {@entry tkBind(event:'<KeyPress-Return>'
                        action:proc {$} {self post} end)}
         {Tk.batch [grid(row:0 column:0 @canvas sticky:ns)
                    grid(row:1 column:0 @entry sticky:ew)
                    grid(row:0 column:1 @vscroll sticky:ns)
                    grid(row:2 column:0 @hscroll sticky:ew)
                    grid(row:3 column:0 @quit sticky:w)
                    grid(columnconfigure self 0 weight:1)
                    grid(rowconfigure self 0 weight:1)]}
      end
      meth show(TEXT)
         {@canvas tk(create text 0 @y text:TEXT anchor:nw tags:@tag)}
         local
            [X1 Y1 X2 Y2] = {@canvas tkReturnListInt(bbox all $)}
         in
            y:=Y2
            {@canvas tk(configure scrollregion:q(X1 Y1 X2 Y2))}
         end
      end
      meth post
         {Port.send @selfPort say({@entry tkReturn(get $)})}
         {@entry tk(delete 0 'end')}
      end
   end
end
```

Part VI

Native C/C++ Extensions

15

Global Counter Library

Oz can be very simply extended with new functionality and datatypes implemented in C or C++. This capability is often used to interface Oz to existing libraries: for example, the regex and gdbm modules are implemented in this fashion.

Every extension is presented to the system in the form of a native functor, i.e. a functor which happens to be implemented in C or C++ rather than in Oz.

In this chapter, we define a native functor that exports a `next` function which returns the next value of a global counter each time it is called.

15.1 Implementation

```
#include "mozart.h"

static long n;

OZ_BI_define(counter_next,0,1)
{
  OZ_RETURN_INT(n++);
}
OZ_BI_end

OZ_C_proc_interface * oz_init_module(void)
{
  static OZ_C_proc_interface table[] = {
    {"next",0,1,counter_next},
    {0,0,0,0}
  };
  n = 1;
  return table;
}
```

`OZ_BI_define(counter_next,0,1)` indicates that we are defining a procedure `counter_next` that implements a new builtin which takes 0 input arguments and returns 1 output value. `OZ_BI_end` is required to finish this definition.

`OZ_RETURN_INT(`*d*`)` is a macro that causes the builtin to return integer *d* as an Oz integer. This should only be used when the builtin has one unique output value; and it is essentially equivalent to the code sequence:

```
OZ_out(0)=OZ_int(d);
return PROCEED;
```

Finally procedure `oz_init_module` implements the native functor: it performs arbitrary initializations and then returns a table of builtins. Each entry in this table consists of (1) the name of the export, (2) the input arity, (3) the output arity, (4) the procedure implementing the builtin. The table must be terminated by an entry whose fields are all zero.

Note that global variable `n` is explicitly initialized by `oz_init_module` rather than with a static initializer. Here, it probably makes no difference, but you cannot in general rely on the fact that constructors for global objects will be properly invoked when the native functor is loaded. What actually happens varies from one system to another. The only reliable technique is to perform all initializations in `oz_init_module`.

You may also define the variable `oz_module_name` to give your native module a name that can be used when printing the builtins which it exports. This is particularly useful for debugging and for interactively looking at values. For example, you could give it the name `"GlobalCounter"`:

```
char oz_module_name[] = "GlobalCounter";
```

15.2 Compilation

We must now make this native module available as a *shared object library*. First we must compile it and create `counter.o`:

```
oztool c++ -c counter.cc
```

Then we must produce a platform specific shared object library:

```
oztool ld counter.o -o counter.so-‘oztool platform‘
```

You may find it useful to create a `Makefile` of the form:

```
PLATFORM = $(shell oztool platform)
NATIVES  = counter
TARGETS  = $(addsuffix .so-$(PLATFORM),$(NATIVES))
all: $(TARGETS)
%.so-$(PLATFORM): %.o
        oztool ld $< -o $@
%.o: %.cc
        oztool c++ -c $< -o $@
```

`oztool` is a program that invokes the facility named as its first argument with appropriate options. For example, it is essential to invoke the same C++ compiler and with the same e.g. code generation options as were used for building the Oz emulator; otherwise it will not be possible to dynamically link your library into a running Oz process. Normally, the Oz emulator is compiled without *run time information* (option `-fno-rtti`

for g++) and without support for C++ exceptions (option `-fno-exceptions` for g++). `oztool c++` automatically invokes the right compiler with the right options. `oztool` is documented in Chapter *The Oz DLL Builder:* `oztool`, *(Oz Shell Utilities)*.

Even more complicated is how to create a DLL from a compiled object file: it varies depending on the system, compiler and linker used. Under Windows, the sequence of necessary incantations is so arcane and highly magical, it could well endanger your sanity. Fortunately `oztool ld` automatically takes care of the details.

15.3 Deployment

Normally, you will then place the resulting shared object file, e.g. `counter.so-linux-i486` on a Linux system, in an installation directory; let's call it *install*. If your site has several platforms sharing one file system, then you can place all platform specific shared object libraries that you create from `counter.cc` in the same *install* directory. They all have distinct names since the platform name is appended.

In an Oz functor, you then write an import of the form:

```
Cnt at 'install/counter.so{native}'
```

The `{native}` suffix indicates to the system that this denotes a native functor whose platform independent basename is *install*`/counter.so`. The module manager dynamically links the appropriate platform specific shared object library (by appending the platform specific extension to the basename) and makes available the module it defines as the value of `Cnt`. The body of your functor can invoke `{Cnt.next}` to get the next value of the global counter.

In the emacs OPI, you can try this out immediately:

```
declare [M] = {Module.link ['install/counter.so{native}']}
```

16

Native Counter Objects

In this chapter, we are going to generalize the counter idea: instead of having just one global counter, we are going to have counter objects implemented as extensions. Of course, this is intended purely as a didactic exercise: such counters could much more easily be defined as Oz objects directly.

16.1 Counter Class

We derive a new `Counter` class from the base class `OZ_Extension`.

```
#include "mozart.h"

class Counter : public OZ_Extension {
public:
  long * n;
  Counter();
  Counter(long*);
  static int id;
  virtual int getIdV();
  virtual OZ_Term typeV();
  virtual OZ_Extension* gCollectV(void);
  virtual OZ_Extension* sCloneV(void);
  virtual void gCollectRecurseV(void) {}
  virtual void sCloneRecurseV(void) {}
  virtual OZ_Term printV(int depth = 10);
};
```

A `Counter` object contains a pointer to a malloced `long`. Why not simply have a member of type `long`: simply because we want to illustrate an application of finalization; malloced memory is a resource that needs to be freed when no longer needed.

The `Counter` class provides implementations for a number of virtual member functions introduced by class `Oz_Extension`. We are now going to explain each of them and provide the necessary code.

16.2 Counter Creation

For this we need the `Counter()` constructor and the new builtin `counter_new`. The constructor allocates a new `long`, sets `n` to its address and initializes it with 1. The builtin creates a new instance of the `Counter` class, boxes it as an Oz value by invoking `OZ_extension` and returns the result.

```
Counter::Counter() { n = new long[1]; n[0]=1; }

OZ_BI_define(counter_new,0,1)
{
  OZ_RETURN(OZ_extension(new Counter));
}
OZ_BI_end
```

16.3 Type Testing

Every extension class should be uniquely identified. This is the purpose of virtual function `getIdV`. Here we illustrate the usual way of doing so: the class is equipped with a static `id` member and `getIdV()` returns it. This static member is initialized by `oz_init_module()` (see Section 16.9).

```
int Counter::id;
int Counter::getIdV() { return id; }
```

Your code will also need to test whether some `OZ_Term` is in fact a boxed `Counter`.

```
inline OZ_Boolean OZ_isCounter(OZ_Term t)
{
  t = OZ_deref(t);
  return OZ_isExtension(t) &&
    OZ_getExtension(t)->getIdV()==Counter::id;
}
```

Additionally, you should probably provide a builtin to perform this test in Oz code:

```
OZ_BI_define(counter_is,1,1)
{
  OZ_declareDetTerm(0,t);
  OZ_RETURN_BOOL(OZ_isCounter(t));
}
OZ_BI_end
```

Finally, it would be nice if `{Value.type C}` would return the atom `counter` when `C` is a counter object.

```
OZ_Term Counter::typeV() { return OZ_atom("counter"); }
```

16.4 Expecting Counter Arguments in Builtins

Obviously we need a way to unbox counter objects.

```
inline Counter* OZ_CounterToC(OZ_Term t)
{
  return (Counter*) OZ_getExtension(OZ_deref(t));
}
```

Now we can define a convenient macro that we can use in the implementation of a builtin to wait until argument `ARG` is determined, check that it is a boxed `Counter`, and declare a variable `VAR` to hold a pointer to its unboxed value.

```
#define OZ_declareCounter(ARG,VAR) \
OZ_declareType(ARG,VAR,Counter*,"counter",OZ_isCounter,OZ_CounterToC)
```

Next, we illustrate how to use this macro.

16.5 Operations on Counters

The first operation obtains the current value of the counter object, but does not change it. We use our new macro to state that the first argument (i.e. argument number 0) should be a determined boxed counter and that `c` should be set to point to its unboxed value.

```
OZ_BI_define(counter_get,1,1)
{
  OZ_declareCounter(0,c);
  OZ_RETURN_INT(*c->n);
}
OZ_BI_end
```

Thanks to our macro, if the argument is not determined, the builtin will automatically suspend, and if it is determined but is not a counter object, it will raise an error exception.

We can similarly define a builtin for setting the value of the counter. It takes 2 arguments: a counter object and an integer.

```
OZ_BI_define(counter_set,2,0)
{
  OZ_declareCounter(0,c);
  OZ_declareInt(1,i);
  *c->n=i;
  return PROCEED;
}
OZ_BI_end
```

Finally, we can define a builtin to obtain the current value of a counter object and post increment the counter by 1.

```
OZ_BI_define(counter_next,1,1)
{
  OZ_declareCounter(0,c);
  long i = *c->n;
  *c->n = i+1;
  OZ_RETURN_INT(i);
}
OZ_BI_end
```

16.6 Printing Support

Of course, it would be nice if `{Show C}`, when `C` is a counter object, would display <counter *n*> where *n* is the current value of the counter. This is easily achieved by defining virtual function `printV` to return an appropriate virtual string.

```
OZ_Term Counter::printV(int depth = 10)
{
  return OZ_mkTupleC("#",3,
                     OZ_atom("<counter "),
                     OZ_int(*n),
                     OZ_atom(">"));
}
```

16.7 Garbage Collection

An instance of an `OZ_Extension` class lives on the heap and must be properly copied at each garbage collection. This is realized simply by creating a new instance (automatically allocated on the *to* heap) and initializing it with the appropriate info. In the case of a counter object, we must copy the `n` pointer. For this purpose we define a one argument constructor.

```
Counter::Counter(long*p):n(p){}
OZ_Extension* Counter::gCollectV() { return new Counter(n); }
```

Cloning is a kind of copying used during search rather than garbage collection. Every variable and every data-structure that has *token* equality (rather than structural equality), e.g. `OZ_Extension`, is *situated* in a space: its home space, i.e. the computation space in which it was created. When its home space H is cloned, the data-structure D must also be cloned: the clone of D must be situated in the clone of H. In the present case, for simplicity we only intend to support counters at top level; thus, the `sClone` method should never be used:

```
OZ_Extension* Counter::sCloneV() { Assert(0); return 0; }
```

16.8 Finalization

When all references to a counter object disappear, we would like the malloced `long` to be freed. We cannot easily register a counter object for finalization from the C++ code (this will have to be delegated to Oz code), but we can provide the implementation of the finalization handler.

```
OZ_BI_define(counter_free,1,0)
{
  OZ_declareCounter(0,c);
  free(c->n);
  return PROCEED;
}
OZ_BI_end
```

16.9 Native Functor

We must now package this library as a native functor. This is done by providing the function `oz_init_module()` which returns a table of builtins. Here, it must also initialize the static member `Counter::id`.

```
OZ_C_proc_interface * oz_init_module(void)
{
  static OZ_C_proc_interface table[] = {
    {"new",0,1,counter_new},
    {"is",1,1,counter_is},
    {"get",1,1,counter_get},
    {"set",2,0,counter_set},
    {"next",1,1,counter_next},
    {"free",1,0,counter_free},
    {0,0,0,0}
  };
  Counter::id = OZ_getUniqueId();
  return table;
}
```

Assuming the code above is put in file `counter-obj.cc`[1], we first compile and then create a DLL as follows

```
oztool c++ -c counter-obj.cc
oztool ld counter-obj.o -o counter-obj.so-'oztool platform'
```

16.10 Oz Wrapper Module

The counter object native library will now be wrapped in an Oz module that registers every new counter object for finalization.

[1] `counter-obj.cc`

```
functor
import
   CNT(new:NEW is:Is get:Get set:Set next:Next free:Free)
   at 'counter-obj.so{native}'
   Finalize(guardian)
export
   New Is Get Set Next
define
   Register = {Finalize.guardian Free}
   proc {New C}
      {NEW C}
      {Register C}
   end
end
```

17

Situated Cell-Like Objects

In this chapter, we implement `Celloid`s: an extension class for objects that behave essentially like cells; they have content which can be accessed (read) and assigned (written). The new challenge here is twofold: (1) during garbage collection the content of a `Celloid` must also be copied, (2) we must ensure that only a local `Celloid` can be mutated (for a non-local one, we should signal an error).

A `Celloid` is *situated*. What does situated mean? Simply that the object resides in one specific constraint store, aka a computation space. If `Celloid C` resides in space S1, and S2 is a subspace of S1 (i.e. is subordinated to S1), it is meaningful for a thread in S2 to *access* `C` since any constraint (therefore value) from S1 are visible in S2, but it is generally not meaningful for a thread in S2 to *assign* `C` since constraints (therefore values) specific to S2 are not visible from S1. Our implementation will enforce the appropriate restrictions.

17.1 Celloid Class

Again, we subclass `OZ_Extension`.

```
#include "mozart.h"

class Celloid : public OZ_Extension {
public:
  OZ_Term content;
  Celloid(OZ_Term t):content(t){}
  static int id;
  virtual int getIdV() { return id; }
  virtual OZ_Term typeV() { return OZ_atom("celloid"); }
  virtual OZ_Extension* gCollectV();
  virtual OZ_Extension* sCloneV();
  virtual void gCollectRecurseV();
  virtual void sCloneRecurseV();
  virtual OZ_Term printV(int depth = 10);
};
```

17.2 Celloid Creation

The `celloid_new` builtin takes one input argument `t`, creates a new celloid whose content is initialized to `t`, boxes it and returns the result.

```
OZ_BI_define(celloid_new,1,1)
{
  OZ_declareTerm(0,t);
  OZ_RETURN(OZ_extension(new Celloid(t)));
}
OZ_BI_end
```

17.3 Type Testing

The definitions here are similar to the ones presented earlier for the `Counter` class.

```
int Celloid::id;

inline OZ_Boolean OZ_isCelloid(OZ_Term t)
{
  t = OZ_deref(t);
  return OZ_isExtension(t) &&
    OZ_getExtension(t)->getIdV()==Celloid::id;
}

OZ_BI_define(celloid_is,1,1)
{
  OZ_declareDetTerm(0,t);
  OZ_RETURN_BOOL(OZ_isCelloid(t));
}
OZ_BI_end
```

17.4 Expecting Celloid Arguments in Builtins

Again this is similar to the `Counter` class: we define an unboxing function and a convenience macro.

```
inline Celloid* OZ_CelloidToC(OZ_Term t)
{
  return (Celloid*) OZ_getExtension(OZ_deref(t));
}

#define OZ_declareCelloid(ARG,VAR) \
OZ_declareType(ARG,VAR,Celloid*,"celloid",OZ_isCelloid,OZ_CelloidToC)
```

17.5 Operations on Celloids

First, we provide an *access* builtin that retrieves the content of the celloid.

```
OZ_BI_define(celloid_access,1,1)
{
  OZ_declareCelloid(0,c);
  OZ_RETURN(c->content);
}
OZ_BI_end
```

Second, we provide an *assign* builtin that sets the content of the celloid. This operation should only be allowed for a thread executing in the home space of the celloid. For a thread executing in a subordinated space, an exception will be raised.

```
OZ_BI_define(celloid_assign,2,0)
{
  OZ_declareCelloid(0,c);
  OZ_declareTerm(1,t);
  if (c->isLocal()) { c->content=t; return PROCEED; }
  else return OZ_raiseErrorC("celloid",3,OZ_atom("nonLocal"),
                             OZ_in(0),OZ_in(1));
}
OZ_BI_end
```

virtual member function `isLocal()` indicates whether the current space is the home space of the celloid. If yes, we set the content to the given argument; if no, we raise an error. `OZ_in(n)` refers to the *n*th input argument of the builtin.

17.6 Printing Support

We provide here only minimal printing support.

```
OZ_Term Celloid::printV(int depth = 10)
{
  return OZ_atom("<celloid>");
}
```

17.7 Garbage Collection

The first part of garbage collection is as before: we create a new instance of `Celloid` initialized with the current content of the celloid that is being copied by gc.

```
OZ_Extension* Celloid::gCollectV() { return new Celloid(content); }
```

The second part involves recursively copying the content of the celloid. This is implemented in a different virtual function:

```
void Celloid::gCollectRecurseV() { OZ_gCollect(&content); }
```

The procedure `OZ_gCollect(OZ_Term*)` performs the gc copy and update of its argument.

You may wonder: why not perform the recursive copy of the content in `gCollectV()` itself. Under no circumstances should you do this! It would break essential invariants in the garbage collector. GC copy must proceed in these 2 phases:

1. `gCollectV()` creates a new instance (on the *to* heap) and initializes it with the current contents of the object being gced.
2. `gCollectRecurseV()` is at some subsequent point invoked on the new instance and should perform the gc copy and update of its contents.

17.8 Cloning

Cloning is used to produce a copy of a computation space. It has the same structure, and the underlying implementation in fact shares most of the code with, garbage collection.

```
OZ_Extension* Celloid::sCloneV() { return new Celloid(content); }
void Celloid::sCloneRecurseV() { OZ_sClone(&content); }
```

17.9 Native Functor

Again, we proceed as before:

```
OZ_C_proc_interface * oz_init_module(void)
{
  static OZ_C_proc_interface table[] = {
    {"new",1,1,celloid_new},
    {"is",1,1,celloid_is},
    {"access",1,1,celloid_access},
    {"assign",2,0,celloid_assign},
    {0,0,0,0}
  };
  Celloid::id = OZ_getUniqueId();
  return table;
}
```

Assuming the code above is put in file `celloid.cc`[1], we first compile and then create a DLL as follows

```
oztool c++ -c celloid.cc
oztool ld celloid.o -o celloid.so-‘oztool platform‘
```

[1] `celloid.cc`

17.10 Oz Wrapper Module

Here, we hardly need an Oz wrapper module. Unlike for counter objects, we don't need to register celloid for finalization: there are no resources off the heap. However, we can provide a nice error print formatter for the case when an access is attempted from without the celloid's home space.

```
functor
import
   Celloid(new:New is:Is access:Access assign:Assign)
   at 'celloid.so{native}'
   Error(registerFormatter)
export
   New Is Access Assign
define
   fun {CelloidFormatter E}
      T = 'Celloid Error'
   in
      case E of celloid(nonLocal C V) then
         error(kind: T
               msg: 'Attempted assign on non local celloid'
               items: [hint(l:'Operation' m:'Celloid.assign')
                       hint(l:'Celloid'   m:oz(C))
                       hint(l:'Value'     m:oz(V))])
      else
         error(kind: T
               items: [line(oz(E))])
      end
   end
   {Error.registerFormatter celloid CelloidFormatter}
end
```

Part VII

Appendices

A

Data and Code Fragments

This appendix contains code fragments left out in the text's chapters.

A.1 Application Development

79a ⟨**Body for DB.oz** 79a⟩≡

```
define
   Data = {Dictionary.new}
   Ctr  = {New class $
                  prop locking
                  attr i:0
                  meth init(I <= 0)
                     lock i := I end
                  end
                  meth get($)
                     lock @i end
                  end
                  meth inc($)
                     lock I=@i+1 in i := I I end
                  end
               end init()}

   proc {Add X}
      I={Ctr inc($)}
   in
      {Dictionary.put Data I X}
   end

   fun {Get ID}
      {Dictionary.get Data ID}
   end

   fun {GetAll}
      {Map {Dictionary.keys Data}
       fun {$ K}
          {AdjoinAt {Dictionary.get Data K} key K}
       end}
```

```
   end

   proc {Remove ID}
      {Dictionary.remove Data ID}
   end
```

80a ⟨Implementation of Book 80a⟩≡

```
T ={New Tk.toplevel tkInit}
F1={New Tk.frame    tkInit(parent:T relief:sunken bd:2)}
V ={New Tk.variable tkInit(Fs.1.key)}
{Tk.batch
 grid(b({Map [’’ ’From’ ’To’ ’Price’]
         fun {$ A}
            {New Tk.label tkInit(parent:F1 text:A
                                 relief:raised bd:1)}
         end})
      padx:1 pady:1 sticky:ew) |
 {Map Fs
  fun {$ F}
     grid({New Tk.radiobutton tkInit(parent:F1 var:V
                                     value:F.key)}
          b({Map [’from’ to price]
             fun {$ A}
                {New Tk.label tkInit(parent:F1 text:F.A)}
             end}))
  end}}
F2={New Tk.frame tkInit(parent:T)}
[FN LN EM] =
{Map [’First name’ ’Last name’ ’E-Mail’]
 fun {$ S}
    E={New Tk.entry tkInit(parent:F2 width:20 bg:wheat)}
 in
    {Tk.send grid({New Tk.label
                   tkInit(parent:F2 text:S#’:’ anchor:w)}
                  E
                  sticky:ew)}
    fun {$}
       {E tkReturnAtom(get $)}
    end
 end}
B={New Tk.button
   tkInit(parent:T text:’Okay’
          action: proc {$}
                     Get=form(first: {FN}
                              last:  {LN}
                              email: {EM}
                              key:   {V tkReturnInt($)})
                     {T tkClose}
                  end)}
```

```
in
{Tk.send pack(F1 F2 B padx:1#m pady:2#m)}
```

81a ⟨**Sample flights** 81a⟩≡

```
[f('from':'Paris'       to:'Stockholm'   price:234)
 f('from':'Saarbrücken' to:'Paris'       price:345)
 f('from':'New York'    to:'Saarbrücken' price:567)
 f('from':'New York'    to:'Bruxelles'   price:363)
 f('from':'Paris'       to:'Saarbrücken' price:834)
 f('from':'Stockholm'   to:'Bruxelles'   price:333)
 f('from':'London'      to:'Saarbrücken' price:523)
 f('from':'Saarbrücken' to:'London'      price:457)
 f('from':'Bruxelles'   to:'New York'    price:324)
 f('from':'Boston'      to:'Stockholm'   price:765)
 f('from':'Stockholm'   to:'New York'    price:344)
 f('from':'Sydney'      to:'Saarbrücken' price:3452)
 f('from':'Sydney'      to:'Stockholm'   price:2568)]
```

Bibliography

[1] Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A Higher-order Module Discipline with Separate Compilation, Dynamic Linking, and Pickling. Technical report, Programming Systems Lab, DFKI and Universität des Saarlandes, 1998. DRAFT.

Index