Oz Shell Utilities

Denys Duchier
Leif Kornstaedt
Christian Schulte

Version 1.3.2
June 15, 2006

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

1

The Oz Engine: `ozengine`

Like Java, Oz is also based on the idea of byte code execution. `ozengine` is an emulator that implements the Oz virtual machine. It is normally invoked as follows:

```
ozengine url args...
```

where *url* identifies an Oz application (a pickled functor) and *args...* are the arguments for the application.

It is also possible to create *executable* functors. These are ordinary functors, except that when one is invoked as a shell command, it automatically starts `ozengine` on itself.

Under Windows, an alternative to `ozengine` is provided called `ozenginew`. While `ozengine` is a console-based application, `ozenginew` is a GUI-based application. Its output appears in message boxes instead of in a console window. Other than that, they behave the same.

Instead of supplying arguments on the command line, you can also indicate a preference for using a graphical interface for editing an application's input parameters:

```
ozengine --gui url
```

The `-gui` option is convenient for starting applications directly from a web browser, simply by clicking. This makes `ozengine -gui %s` a good choice of a helper application for web documents with MIME type `application/x-oz-application`. Consult Chapter *Application Deployment, (Application Programming)* for more detailed information.

2

The Oz Compiler: ozc

The Oz compiler is extensively documented in *"The Mozart Compiler"*. We describe it here in its incarnation as a command line application. It is most frequently invoked as follows:

```
ozc -c Foo.oz
```

File `Foo.oz` is expected to contain an expression which is then compiled and evaluated and the resulting value is written (pickled) into file `Foo.ozf`.

2.1 Batch Compiler Modes

The batch compiler can operate in one of several mutually exclusive modes selected by the options below:

`-h`, `-?`, `-help`

Prints out information on all legal options, then exits.

`-c`, `-dump`

`ozc -c ...` *FILE*`.oz ...`

Compiles and evaluates the expression in file *FILE*`.oz`, then pickles the resulting value into the output file, *FILE*`.ozf` by default.

`-e`, `-execute`

`ozc -e ...` *FILE*`.oz ...`

Compiles and executes the statement in file *FILE*`.oz`. This is the default mode of operation.

`-E`, `-core`

`ozc -E ...` *FILE*`.oz ...`

Translates the statement in file *FILE*`.oz` into the core language, then writes this expanded form into the output file, *FILE*`.ozi` by default.

`-S`, `-scode`

`ozc -S ...` *FILE*`.oz ...`

Compiles the statement in file *FILE*`.oz` into the assembly bytecode format, then writes it to the output file, *FILE*`.ozm` by default.

-s, **-ecode**

`ozc -s ...` *FILE*`.oz ...`

Like `-scode`, except that the file is compiled as an expression instead of as a statement.

-x, **-executable**

`ozc -x ...` *FILE*`.oz ...`

Much like `-c`, compiles and evaluates the expression in file *FILE*`.oz`, then writes the resulting value into the output file, by default *FILE* (without extension) under Unix or *FILE*`.exe` under Windows. Additionally, it makes the output file executable. Thus, if `Foo.oz` contains an application functor,

```
ozc -x Foo.oz
```

creates the file `Foo` (`Foo.exe` under Windows), which is executable and can be invoked directly from the shell. Note also that you can always run an application functor as follows:

`ozengine` *url args* `...`

where *url* is a URL or pathname that references the application functor.

2.2 Options Valid in All Modes

-v, **-verbose**

Display all compiler messages.

-q, **-quiet**

Inhibit compiler messages.

-M, **-makedepend**

Instead of executing, write a list of dependencies to stdout.

2.3 Options for Output Files

-o *FILE*, **-outputfile=***FILE*

Write output to *FILE* (`-` for stdout). If this option is given, then a single input file may be given. Otherwise, an arbitrary number of input files can be processed.

-unix, **-target=unix**

When invoked with `-x`, produce Unix executables even if running under Windows.

-windows, **-target=windows**

When invoked with `-x`, produce Windows executables even if running under Unix.

-execheader=*STRING*

When invoked with option `-x`, the compiler prepends a header to the output file so that it is interpreted as executable by the operating system. Under Unix, the default header is as follows:

```
#!/bin/sh
exec ozengine $0 "$@"
```

The `-execheader` option allows you to specify a different header.

`-execpath=`*STRING*

Uses the header given above in the description of `-execheader`, except that `ozengine` is replaced by *STRING*.

`-execfile=`*FILE*

Reads in *FILE* and uses this as header. Under Windows, the default behaviour is to use the file provided in *ozhome*`/bin/ozwrapper.bin`, where *ozhome* is Mozart's installation folder. `ozwrapper.bin` is a Windows executable that launches `ozengine`, i.e., a console application (CUI).

`-execwrapper=`*FILE*

Reads in *ozhome*`/bin/`*FILE* and uses it as header for executable files. Apart from `ozwrapper.bin`, an `ozwrapperw.bin` is supplied that launches `ozenginew` instead of `ozengine`, i.e., a Windows application (GUI). This option provides a convenient way to use this alternative wrapper.

`-z` *N*, **`-compress=`***N*

Pickles may be written in a compressed format. By default they are not compressed. `-z` *N* selects the compression level: *N* is an integer between 0 (uncompressed) and 9 (maximum compression). Compressing a pickle may improve loading/downloading time.

2.4 Options to Set the Compiler State

For the following options, the order is important, and even in which order they are intermixed with the input files: When an input file is processed, the compiler state is determined by all options preceding it. Options may be overridden by other options given later on the command line.

Macro Directives

`-D` *NAME*, **`-define=`***NAME*

Define macro *NAME*. Macros allow for conditional compilation using `\ifdef` *NAME* and `\ifndef` *NAME* macro directives.

`-U` *NAME*, **`-undefine=`***NAME*

Undefines macro *NAME*.

Environment

`-l` *MODULES*, **`-environment=`***MODULES*

Makes *MODULES*, a comma-separated list of pairs *VAR*=*URL*, available in the environment. For each *VAR*=*URL*, the module available through the functor at *URL* is obtained and *VAR* is bound to it. The source files then compiled can reference variable *VAR*.

Inserting Files

-I *DIR*, **-incdir=***DIR*

Adds *DIR* to the head of OZPATH which is used to locate files to `\insert`.

-include=*FILE*

Compile and execute the statement in *FILE* before processing the remaining options. For instance, this can be used to extend the compilation environment by executing a `declare`.

Compiler Switches

Most of the compiler switches can be set via command line options. Please refer to Appendix *Compiler Switches, (The Mozart Compiler)* for more detailed descriptions and defaults.

-g, **-(no)debuginfo**

Emits code with debugging information. Use this option if you want to use the Mozart Debugger[1]. The `-g` option is actually an abbreviation for the combination of `-controlflowinfo` and `-staticvarnames`.

-(no)controlflowinfo

Include control flow information.

-(no)staticvarnames

Include static variable name information.

-(no)dynamicvarnames

Attach print names to variables created at run-time.

-p, **-(no)profile**

Emits code with profiling information. Use this option if you want to use the Mozart Profiler[2].

-(no)gump

Allow Gump definitions.

-(no)compilerpasses

Show compiler passes.

-(no)warnredecl

Warn about top-level redeclarations.

-(no)warnshadow

Warn about all redeclarations.

[1] *"The Mozart Debugger"*
[2] *"The Mozart Profiler"*

-(no)warnunused

Warn about unused variables.

-(no)warnunusedformals

Warn about unused variables and formals.

-(no)warnforward

Warn about forward class declarations.

-(no)warnopt

Warn about missed optimizations.

-(no)expression

Expect expressions, not statements.

-(no)allowdeprecated

Allow use of deprecated syntax.

-(no)staticanalysis

Run static analysis.

-(no)realcore

Output the real non-fancy core syntax.

-(no)debugvalue

Annotate variable values in core output.

-(no)debugtype

Annotate variable types in core output.

Compiler Options

-maxerrors=*N*

Limit the number of errors reported to *N*.

-baseurl=*STRING*

Set the base URL to resolve imports of computed functors to *STRING*.

-gumpdirectory=*STRING*

Set the directory where Gump output files are placed to *STRING*.

3

The Oz Linker: `ozl`

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:

- Installing the application requires correct installation of a large number of functors.
- 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 hierarchy 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 excutes the toplevel application functor together with all included functors.

A short introduction by means of examples can be found in Section *Linking Functors, (Application Programming)*.

3.1 Basic Usage

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

```
% ozl In -o Out
```

Consider for example the pickled functors `A.ozf`, `B.ozf`, and `subdir/C.ozf`, where `A.ozf` has been created from the following functor definition:

```
functor
import B
       C at 'subdir/C.ozf'
       Application
end
```

and the other functors have empty imports. By executing

```
% ozl A.ozf -o D.ozf
```

a new pickled functor `D.ozf` is created that contains both the functors contained in `B.ozf` and `subdir/C.ozf` but not the system functor `Application`.

If the linker is invoked in verbose mode as

```
% ozl --verbose A.ozf -o D.ozf
```

or

```
% ozl -v A.ozf -o D.ozf
```

for short, it prints the following information on which functors are in fact included and which are still imported by the newly created functor.

```
Include:
   A.ozf, B.ozf, subdir/C.ozf.
Import:
   x-oz://system/Application.
```

If we now invoke the linker on the newly created pickled functor `D.ozf` in verbose mode as follows:

```
% ozl -v D.ozf
```

it only prints the following information without creating a new functor:

```
Include:
   D.ozf.
Import:
   x-oz://system/Application.
```

By default, the linker includes (or links) all functors that are referred to by relative urls as in our previous example. How to change this behaviour is discussed in Section 3.2.

3.2 Including and Excluding Functors

`-relative`

`-relative` (default), `-norelative`

All functors that are referred to by relative urls are included. Import urls in the resulting functor remain relative.

`-include`

`-include=URL,...,URL`

Include all functors whose url matches one of the comma separated url prefixes.

`-exclude`

`-exclude=URL,...,URL`

Exclude all functors whose url matches one of the comma separated url prefixes.

`-rewrite`

`-rewrite=RULE,...,RULE`

When the functors are gathered that make up the linked functor, all import URLs are resolved with respect to the importing functor. This means that all file urls become absolute, and as such make up the import of the output functor.

This is often not desirable. In the common case that the imported functors lie in the same directory or in subdirectories as the root functor, the `-relative` option can be used, in which case the import urls remain relative.

The `-rewrite` option generalizes this principle: It allows to specify a list of rules of the form `FROM=TO`, meaning: If a url has prefix `FROM`, then replace it by `TO`. The first matching rule is applied.

Note that multiple `-include` and `-exclude` directives can be given on the commandline. They have cummulative effect, with the policy that later directives take precedence over earlier ones. For example:

```
ozl --include=/foo/ --exclude=/foo/bar/ ...
```

causes all imports from files below directory `/foo` to be included except those under directory `/foo/bar`. We can further refine this policy by introducing an exception to the last exclusion pattern and request inclusion of modules imported from below directory `/foo/bar/baz`:

```
ozl --include=/foo/ --exclude=/foo/bar/ --include=/foo/bar/baz/ ...
```

3.3 Pickling Options

The linker supports the following default options for pickles.

`-compress`

`-compress=N`, `-z N`

The created pickle is compressed with level `N` (a single digit). By default the compression level `N` is `0`, that is, no compression is employed.

`-executable`

`-executable` (`-x`), `-noexecutable` (default)

Output the pickled functor as being executable (that is, with an additional header).

`-execheader=`*STRING*

When invoked with option `-x` the linker first outputs a header so that the output file may be interpreted by the operating system as an executable. Under Unix, the default behaviour is to use the following as header:

```
#!/bin/sh
exec ozengine $0 "$@"
```

The `-execheader` option allows you to specify a different header.

`-execpath=`*STRING*

Uses the header given above in the description of `-execheader`, except that `ozengine` is replaced by *STRING*.

`-execfile=`*FILE*

Reads in *FILE* and uses this as header. Under Windows, the default behaviour is to use the file provided in *ozhome*`/bin/ozwrapper.bin`, where *ozhome* is Mozart's installation folder. `ozwrapper.bin` is a Windows executable that launches `ozengine`.

`-execwrapper=`*FILE*

Reads in *ozhome*`/bin/`*FILE* and uses it as header for executable files. Apart from `ozwrapper.bin`, an `ozwrapperw.bin` is supplied that launches `ozenginew` instead of `ozengine`. This option provides a convenient way to use this alternative wrapper.

`-target=unix`, **`-unix`**
`-target=windows`, **`-windows`**

When creating an executable functor, do it for the specified target platform rather than for the current host platform.

3.4 Miscellaneous Options

`-sequential`

`-sequential`, `-nosequential` (default)

Do not create a thread per executed functor body, rather execute all functor bodies in the same thread in a bottom up fashion.

Use with care! In case the functors included have cyclic imports, it is not used.

4

The Oz Debugger: `ozd`

The Oz debugger is extensively documented in *"The Mozart Debugger"*. We describe it here merely in its incarnation as a command line application. Furthermore, we only document its options.

If you have created an Oz application which you normally start from the shell as follows:

```
Foo Args ...
```

Then you can run it under control of the Oz debugger by using the following command instead:

```
ozd Foo -- Args ...
```

Any Oz application can be run in the debugger, but you only get the full benefit of the debugging interface when the code being executed was compiled with the `-g` option to include debugging information.

The double dash `-` separates the arguments intended for `ozd` from those intended for the application being run under the debugger.

`-help`, `-h`, `-?`

Display information on legal options, then exit

`-g`, `-debugger`, `-mode=debugger`

This option is the default: it starts the debugger. The other possibility is `-p` to start the profiler (see Chapter 5).

`-r`, `-remotedebugger`, `-mode=remotedebugger`

In this mode of operation, a debugger client is started and connects to a debugger server. The `-ticket` option is required.

`-p`, `-profiler`, `-mode=profiler`

This is the other mode of operation: it starts the profiler instead (see Chapter 5).

`-E`, `-(no)useemacs`

Starts a subordinate Emacs process. This will be used to display the source code currently being debugged. You will also be able to set breakpoints easily on source lines.

`-emacs=`*FILE*

Specifies the Emacs binary to run for option `-E`. The default is $`OZEMACS` if set, else `emacs`.

`-ticket=`*TICKET*

Specifies the ticket to use for option `-r` to connect to the server.

5

The Oz Profiler: `ozd -p`

The Oz profiler is extensively documented in *"The Mozart Profiler"*. We describe it here merely in its incarnation as a command line application. Furthermore, we only document its options.

If you have created an Oz application which you normally start from the shell as follows:

```
Foo Args ...
```

Then you can run it under control of the Oz profiler by using the following command instead:

```
ozd -p Foo -- Args ...
```

Any Oz application can be run in the profiler, but you only get the full benefit of the profiling interface when the code being executed was compiled with the `-p` option to include profiling instrumentation code. The profiler and the debugger share the same interface.

The double dash `-` separates the arguments intended for `ozd` from those intended for the application being run under the profiler.

`-help`, `-h`, `-?`

Display information on legal options, then exit

`-p`, `-profiler`, `-mode=profiler`

You must supply this option in order to start the profiler; otherwise the debugger is started instead (see Chapter 4).

`-g`, `-debugger`, `-mode=debugger`

This is the default option: it starts the debugger (see Chapter 4). As mentioned above, in order to actually start the profiler, you must supply the `-p` option.

`-E`, `-(no)useemacs`

Starts a subordinate Emacs process. This will be used to display the source code corresponding to the profile data being examined.

`-emacs=`*FILE*

Specifies the Emacs binary to run for option `-E`. The default is `$OZEMACS` if set, else `emacs`.

6

The Oz DLL Builder: `oztool`

`oztool` facilitates the creation of native functors (see Part *Native C/C++ Extensions, (Application Programming)* and *"Interfacing to C and C++"*). A native functor is a DLL, i.e. a library that is dynamically loaded by the Oz emulator and interfaces with it. Creating a native functor often involves complicated compilation and linking technicalities (e.g. options). `oztool` takes care of these details for you.

`oztool c++ ...`

Instead of calling the C++ compiler directly, you should invoke it through `oztool`. The advantages are: it calls the right compiler, with the right options, and also extends the include search path to find the Mozart specific includes such as `mozart.h`. Normally, you would compile a native functor implemented in `foo.cc` using:

```
oztool c++ -c foo.cc
```

`oztool cc ...`

Same idea, but for the C compiler

`oztool ld ...`

Instead of calling the linker directly, you should also invoke it through `oztool`. Again, the advantages are that it calls the right linker, with the right options. Normally, you would create a DLL from `foo.o` as follows:

```
oztool ld -o foo.so foo.o
```

`oztool platform`

The default Resolution[1] mechanism locates architecture specific DLLs as follows: If the system needs a native functor called (abstractly) `foo.so`, then it will look for a DLL called `foo.so-linux-i486` on a Linux machine, `foo.so-solaris-sparc` on a Solaris machine, or `foo.so-win32` on a Windows machine, etc... Thus, when you create a DLL, you should install it with a name where the machine's architecture is appended. Invoking `oztool platform` simply prints out the appropriate architecture name for your machine. In this respect, `oztool` helps you write portable Makefiles: to create a DLL from file `foo.cc` you would normally invoke:

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

[1] Chapter *Resolving URLs:* `Resolve`, *(System Modules)*

6.1 Windows Environment Variables

Under Windows, `oztool` supports the use of several compilers as described in Chapter *Creating DLLs under Windows, (Interfacing to C and C++)*. Furthermore, when using the GNU compiler suite, the exact name of the compilers and linker to invoke can be set by the environment variables shown in the following table with their default values.

Environment Variable	Default Value
OZTOOL_CC	`gcc -mno-cygwin`
OZTOOL_CXX	`g++ -mno-cygwin`
OZTOOL_LD	`g++ -mno-cygwin`

These values correspond to the compilers used to build Mozart. The above binaries are provided by the Cygwin `gcc` package. Note that using a different compiler can lead to problems, as described in Section *Known Bugs and Problems, (Interfacing to C and C++)*.

7

Conversion of Pickles: `convertTextPickle`

If you had saved data structures into pickles with Mozart 1.0.1, then you'll want to use them in newer releases of Mozart as well. The format of pickles has changed after 1.0.1, however. This command line utility can help you to convert pickles. Concerning limitations and how to overcome them, please refer to the compat[1] module.

`-help`, **`-h`**, **`-?`**

Display information on legal options, then exit.

`-in=`*file*, **`-i`** *file*

Select the file from which to read a text pickle in old format.

`-out=`*file*, **`-o`** *file*

Select the file to which to write the converted pickle in new format.

[1] Chapter *Backwards Compatibility, (Contributed Libraries)*