# Interfacing to C and C++

Michael Mehl
Tobias Müller
Christian Schulte
Ralf Scheidhauer

Version 1.3.2
June 15, 2006

## Abstract

Oz provides a simple yet powerful interface to dynamically link native C and C++ code to Oz. It provides for access and conversion from most Oz values to C data structures and vice versa and supports mechanisms to handle suspension for C(++) functions.

## Credits

Mozart logo by Christian Lindig

## License Agreement

# Contents

# 1

# Introduction

Oz provides a simple yet powerful interface to dynamically link native C and C++ code to Oz[1]. It provides for access and conversion from most Oz values to C data structures and vice versa and supports mechanisms to handle suspension for C(++) functions.

The usage of the C and C++ interface is first explained using an example followed by a reference part that describes the interface in detail.

[1]Dynamic linking is currently not supported on the AIX/RS6000 platform.

# 2

# A small Example

**Figure 2.1** File `getenv.cc`: a C++ program to provide a getenv for Oz

```
#include "mozart.h"

OZ_BI_define(BIgetenv,1,1)
{
  OZ_declareAtom(0,envVarName);

  char *envValue = getenv(envVarName);

  if (envValue == 0) /* not defined in environment */
    return OZ_FAILED;

  OZ_RETURN_ATOM(envValue);
} OZ_BI_end


static OZ_C_proc_interface oz_interface[] = {
  {"getenv",1,1,BIgetenv},
  {0,0,0,0}
};

OZ_C_proc_interface *oz_init_module() {
  return oz_interface;
}
```

Suppose we want to provide an Oz native module `Goodies` containing a single procedure `{Goodies.getenv VarA ValueA}` as an interface to the C library function `getenv(3)`: it constrains `ValueA` to an atom, which is the value of the environment variable `VarA`, where `VarA` is an atom. Thus `{Goodies.getenv 'HOME' X}` will constrain `X` to an atom representing the path to our home directory. To realize this we have to perform the following steps:

1. Write a piece of C code.
2. Create an object file from the C code.

3. Create a dynamic library from the object file(s).
4. Link the library into Oz.

These steps are explained in more detail below.

# 3

# The C Part

Figure 2.1 shows a first attempt to provide access to the `getenv` C library function as explained above. In the following we will go through it in detail and successively improve it.

Every C program that will be linked to Oz must include the header file `mozart.h` (line 1 in Figure 2.1), which is located in the `include` subdirectory of the Oz installation directory. It contains the definition of the data structures and functions used to interface C to Oz. All these data structures start with `OZ_` such that they will not clash with the user's name space.

To declare a C function that can be used from Oz you have to enclose its declaration into the macros `OZ_BI_define(`*name*`,`*inarity*`,`*outarity*`)` and `OZ_BI_end` . The following code fragment declares a C-function `BIgetenv` for inclusion into Oz, which has one input an one output argument:

```
OZ_BI_define(BIgetenv,1,1)
{
  ...
} OZ_BI_end
```

Oz represents data like strings, integers, floats, etc. different than C. For this reason `mozart.h` provides type testing functions and routines to convert from the C into the Oz representation and vice versa. All Oz values are summarized in one abstract C data type called `OZ_Term` .

To signal whether the call to a C function was successful or not it must return a value of type `OZ_Return`, which may be `OZ_FAILED` (in which case failure will occur), `OZ_ENTAILED` to signal successful completion. `OZ_Return` also contains several other values, which are not visible to the user. They are only used for internal purposes, for example to handle suspension or raising exceptions.

It is important that your C function returns one of the above values. Not returning a value will inevitably crash the executing Oz engine!

In line 5 we use the macro `OZ_declareAtom(n,name)`: it checks whether the `n`-th input argument is an atom and declares a new C++ variable of type `char*` with name `name`. So line 5 declares `envVarName` which holds the C++ string representation of the first and only input argument.

Values that can be passed as input arguments can of course be logic variables! Do not confuse this with output arguments: Output arguments are handled only after the C function returns!

In line 7 we declare a C++ variable `envValue` which will temporarily hold the return value of `BIgetenv`.

In lines 9–11 of Figure 2.1 we check whether an environment variable of the requested name exists and return `OZ_FAILED` if not.

Line 19 loads the result into the output argument of `BIgetenv` using the macro `OZ_RETURN_ATOM`: it converts its argument to an Oz atom assigns the first output register to that value and leaves the function with `OZ_ENTAILED` signalling that the call to `BIgetenv` was success full.

Lines 16–23 in Figure 2.1 are needed for the linking step and will be explained in Section 4.3.

# 4

# Creating a DLL

## 4.1 Compiling the C++ program

The Mozart system provides oztool[1] which we recommend you invoke instead of calling the C/C++ compiler directly:

```
oztool c++ -c getenv.cc -o getenv.o
```

oztool takes care of many unpleasant details for you; for example, it supplies the compiler with the appropriate option for the generation of position independent code.

## 4.2 Creating a dynamic library

Now, we create a shared object from the compiled object file obtained above. Again you should invoke oztool rather than call the linker directly:

```
oztool ld getenv.o -o getenv.so
```

This takes care of many ugly details (especially on Windows where they could easily drive you nuts). Actually, you should really create a shared object file with, as suffix, the platform for which it was created. For example:

```
oztool ld getenv.o -o getenv.so-linux-i486
```

The reason is that the Oz module manager was designed to support platform independent module import specifications, but, of course, native modules must be resolved to platform dependent implementations. The default resolution strategy achieves this by means of platform suffixes.

In order to write portable makefiles, you can use oztool to print out the platform name:

```
oztool platform
```

Thus a portable way to create the shared object is:

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

[1] Chapter *The Oz DLL Builder:* `oztool`, *(Oz Shell Utilities)*

## 4.3 Linking a native module

In the last step we make the native module available by linking it into Oz. Lines 16–23 in Figure 2.1 are needed to declare the export signature of the native module. A function named `oz_init_module` must be exported by every native module: this function will be called when the module is linked into Oz. It can be used to do some module dependent initialization and has to return an array whose elements are of type `OZ_C_proc_interface`; the end of the array must be terminated by an empty structure. The array describes the signature of the functions being exported from the module:

```
typedef struct {
  const char * name;
  short        inArity;
  short        outArity;
  OZ_CFun      func;
} OZ_C_proc_interface;
```

`name` is a string naming the feature under which the native function will be accessible from Oz (see below). `inArity` and `outArity` specify the number of input and output arguments. `func` is a pointer to the function being exported.

Now we can link our module into Oz by executing:

```
declare
[Goodies]={Module.link ['./goodies.so{native}']}
```

This will lazily load the module upon first access and bind `Goodies` to a record with a single (since we exported only one function) feature named `getenv` (this is derived from the value of the `name` field of `OZ_C_proc_interface`). Now we can call it like this:

```
{Browse {Goodies.getenv 'HOME'}}
```

The module can also be imported by any Oz module:

```
functor
import G at 'getenv.so{native}'
define
   ... {G.getenv ...} ...
end
```

Note that the url used in the import specification does not supply the platform suffix, but adds the `{native}` annotation. The module manager (or more precisely the resolver) will remove the annotation and replace it with the suffix appropriate for the current platform.

# 5

# Creating DLLs under Windows

`oztool` works under Windows as described in Chapter 4. In addition `oztool` can be directed to call different compilers and linkers to create object files and DLLs. This is done by specifying one of the following options to `oztool` (`-gnu` is the default):

- `-gnu`: use mingw32[1] or cygwin (i.e., egcs)
- `-msvc`: use Microsoft Visual C++ 5.0/6.0
- `-watcom`: use Watcom 10.x

You can use the option `-verbose` to let `oztool` print out the commands it executes. This will also give you an idea of what steps to perform if you want to use another compiler than those mentioned in the above list.

## 5.1 Known Bugs and Problems

1. When creating DLLs with cygwin[2], make sure that `cygwin.dll` is correctly initialized. The document README.jni.txt[3] might give some more help under `"Cygwin notes"`.
2. It is not possible to link to a C run-time library other than `msvcrt.dll`.
3. Because of incompatible name-mangling schemes of different C++ compilers, a library making use of the foreign function interface's C++ classes must be compiled with GNU C++ to link correctly.

[1] http://www.xraylith.wisc.edu/~khan/software/gnu-win32/index.html
[2] http://sourceware.cygnus.com/cygwin/
[3] http://www.xraylith.wisc.edu/~khan/software/gnu-win32/README.jni.txt

# 6

# Tuning the Example

**Raising Exceptions** In line 10 the program simply returns `OZ_FAILED` if the environment variable is not defined, which is not good programming style. It should better raise an exception. This can be done using

```
OZ_Return OZ_raise(OZ_Term t);
```

which raises the exception `t`. In our example we should replace line 10 with something like

```
return OZ_raise(OZ_atom("envVarNotDefined"));
```

We leave as an exercise to the reader to give more informative exception, e.g. adding the name of the undefined variable.

**Raising type errors** Furthermore an extra function is provided for raising type errors. The macro `OZ_declareAtom` used in our example makes use of this function. Type errors can be signaled using

```
OZ_Return OZ_typeError(int pos, char *expectedType);
```

This is an exception signaling that the argument at position `pos` is incorrect and the name of the expected type is `expectedType`.

**Suspension of C functions** The macro `OZ_declareAtom` internally also makes use of facilities that allow C functions to suspend the running thread on variables. Thus `OZ_declareAtom` uses some code of the following form:

```
if (OZ_isVariable(envVarName)) {
  OZ_suspendOn(envVarName);
}
```

If `envVarName` is an unconstrained variable then `OZ_suspendOn` is called. `OZ_suspendOn` is a macro that takes a variable as argument and suspends the current thread. If the variable is determined the suspended thread becomes runnable in which case it will reexecute the C function *from the beginning*.

The application

```
declare X in {Browse {Goodies.getenv X}}
```

will call the C function as above. But the first argument is detected as variable and the executing thread suspends.

If we feed

```
X='HOME'
```

the C function `BIgetenv` is called again from the beginning and the browser updates the display of the value of the environment variable as expected.

# 7

# Specification of the C interface

Below we give a reference of the functionality provided by the C and C++ interface.

## 7.1 General Remarks

Before we go into more detail we start with some general remarks.

Several functions of the interface rely on the fact that their arguments are of a certain type. For example

```
char *OZ_atomToC(OZ_Term t)
```

expects t to be an Oz atom and returns a string representing the print name of t. In case t is not an atom, the behaviour of OZ_atomToC is undefined and it will typically crash the whole system.

If not stated otherwise all functions that return pointers into memory, return a pointer to a memory area that is allocated *statically*. This provides highest flexibility and efficiency. For example the string returned by OZ_atomToC must not be overwritten by the user and the next call to any of the interface functions may modify it. So the user should take care to make a copy of these memory blocks if necessary and free it again himself.

## 7.2 Data types

The following data types are defined in the interface:

**OZ_Return**

```
typedef enum {OZ_FAILED=0, OZ_ENTAILED=1, ... } OZ_Return;
```

Return values for C functions to be interfaced to Oz.

**OZ_CFun**

```
typedef OZ_Return (*OZ_CFun)(OZ_Term * [])
```

Signature of an interface function.

**OZ_Term**

```
typedef ... OZ_Term;
```

This is the *abstract* data type for Oz values.

**OZ_C_proc_interface**

```
typedef struct {
  const char * name;
  short        inArity;
  short        outArity;
  OZ_CFun      func;
} OZ_C_proc_interface;
```

This structure declares the signature of a function being exported from a native module.

## 7.3 Declaration

**OZ_BI_define**
**OZ_BI_end**

```
OZ_BI_define(Name,InArity,OutArity)
{ ... C/C++ code ... }
OZ_BI_end
```

Every foreign function imported to Oz has to be declared using this pattern. `Name` is the name of the function being defined. `InArity` and `OutArity` specify the number of input and output arguments the function expects.

## 7.4 Accessing arguments

### 7.4.1 Accessing input arguments

**`OZ_declareTerm(n,var)`**

Declares a new variable of type `OZ_Term` named `var`, which is initialized with the value of the `n`-th (counting starts from zero) input argument.

In case you plan to use unification for passing output arguments, you still have to pass the logic variable with which you want to unify as *input* argument.

**`OZ_declareDetTerm(n,var)`**

Works like `OZ_declareTerm` but additionally suspends if the input argument is a free variable.

**`OZ_declareInt(n,var)`**

The function expects in input argument number `n` an Oz integer. It then declares a variable named `var` of type `int` and initializes `var` with the value of this argument. The macro raises an exception if the argument is ill typed and suspends if the argument is an unbound variable.

**`OZ_declareFloat(n,var)`**

Works like `OZ_declareInt` but expectes an Oz float and declares a variable of type `double`.

**OZ_declareAtom(n,var)**

Works like `OZ_declareInt` but expectes an Oz atom and declares a variable of type `char *`.

**OZ_declareVirtualString(n,var)**

Works like `OZ_declareInt` but expectes an Oz virtual string and declares a variable of type `char *`.

**OZ_declareVS(n,var,len)**

Like `OZ_declareVirtualString`, but additionally sets `len` to the size of the result.

**OZ_declareBool(n,var)**

Declares a variable of type `int` named `var`, which is non-zero iff the `n`-th argument is equal to `true`.

The above macros always declare a new C variable and then do some checks. Therefor in C (not in C++) only one of them can be used only at the start of a new block statement. For this reason there is also a second set of macros named `OZ_set*` that expect that their second argument has already been declared. Thus in C++ you can use

```
OZ_declareAtom(0,mystring);
OZ_declareInt(1,myint);
```

whereas in plain C you have to write

```
char *mystring;
int myint;
OZ_setAtom(0,mystring);
OZ_setInt(1,myint);
```

### 7.4.2 Accessing output arguments

**OZ_out(n)**

Abstract access to output argument number `n` (counting starts with 0). Should only be used for writing an output argument and never for reading. Usage is like

```
OZ_out(3) = OZ_atom("myResult");
```

This macro should only be used in case a function returns more than one value. For returning values in the first output argument one of the functions below should be used.

**OZ_RETURN(V)**

Returns from the C function with output value `V`. It is a macro which expands to

```
return (OZ_out(0)=V,OZ_ENTAILED)
```

For convenience we also provide the following macros:

**OZ_RETURN_INT(I)**

Return a C integer. Expands to `OZ_RETURN(OZ_int(I))`

**`OZ_RETURN_ATOM(A)`**

Return a C integer. Expands to `OZ_RETURN(OZ_atom(A))`

**`OZ_RETURN_STRING(S)`**

Return a C integer. Expands to `OZ_RETURN(OZ_string(S))`

**`OZ_RETURN_BOOL(X)`**

Returns `false` if X equals to 0, `true` otherwise. Expands to `OZ_RETURN((X)?OZ_true():OZ_false()`

## 7.5 Type testing

To check whether a given `OZ_Term` is a certain Oz value several functions are provided:

**`OZ_isAtom`**
**`OZ_isBool`**
**`OZ_isCell`**
**`OZ_isThread`**
**`OZ_isPort`**
**`OZ_isChunk`**
**`OZ_isDictionary`**
**`OZ_isCons`**
**`OZ_isFalse`**
**`OZ_isFeature`**
**`OZ_isFloat`**
**`OZ_isInt`**
**`OZ_isBigInt`**
**`OZ_isSmallInt`**
**`OZ_isNumber`**
**`OZ_isLiteral`**
**`OZ_isName`**
**`OZ_isNil`**
**`OZ_isObject`**
**`OZ_isPair`**
**`OZ_isPair2`**
**`OZ_isProcedure`**
**`OZ_isRecord`**
**`OZ_isTrue`**
**`OZ_isTuple`**
**`OZ_isUnit`**
**`OZ_isValue`**
**`OZ_isVariable`**
**`OZ_isBitString`**
**`OZ_isByteString`**
**`OZ_isFSetValue`**

All these functions have the same signature. For example `OZ_isAtom` is declared as `int OZ_isAtom(OZ_Term t)`

All type tests return nonzero iff their argument is of the respective type.

A few of these need some more explanation:

`int OZ_isBigInt(OZ_Term t) int OZ_isSmallInt(OZ_Term t)`

The emulator has two representations for integers: small integers and big integers. Small integers are implemented very efficiently.

`int OZ_isPair(OZ_Term t)`

Returns zero iff `t` is a tuple with label `#`.

`int OZ_isPair2(OZ_Term t)`

Returns nonzero iff `t` is a tuple with label `#` and arity of 2.

`int OZ_isValue(OZ_Term t)`

Returns nonzero iff `t` is not a variable.

`int OZ_isVariable(OZ_Term t)`

Returns nonzero iff `t` is a variable.

**`OZ_isList`**

```
int OZ_isList(OZ_Term term, OZ_Term *var)
```

Returns nonzero iff `term` is a list. If `term` is no list, but the tail is a variable, then `*var` is set to the tail of the list, else it is set to null. `var` may be null. If `term` is cyclic then `OZ_isList` never terminates!

**`OZ_isString`**

```
int OZ_isString(OZ_Term term, OZ_Term *var)
```

Returns nonzero iff `term` is an Oz string. If `term` is no string, but the tail or an element of the list is a variable, then `*var` is set to this variable, else it is set to null. `var` may be null. If `term` is cyclic then `OZ_isString` never terminates!

**`OZ_isVirtualString`**

```
int OZ_isVirtualString(OZ_Term term, OZ_Term *var)
```

Returns nonzero iff `term` is a virtual string. If `term` is no virtual string, but contains a variable, then `*var` is set to this variable, else it is set to null. `var` may be null. If `term` is cyclic then `OZ_isVirtualString` never terminates!

**`OZ_termType`**

```
OZ_Term OZ_termType(OZ_Term t)
```

Returns an atom describing the type of `t`. The following types are returned:

```
variable, int, float, atom, name, tuple, record, fset,
foreignPointer, procedure, cell, space, object, port,
chunk, array, dictionary, lock class, resource
```

(see also `Value.type` in Section *Variable Status*, *(The Oz Base Environment)*).

## 7.6 Conversion

The following functions are used to convert from Oz values to C data structures and vice versa.

**OZ_atom**

```
OZ_Term OZ_atom(char *s)
```

Converts C string `s` to an Oz atom.

**OZ_atomToC**

```
char *OZ_atomToC(OZ_Term t)
```

Converts Oz atom `t` to a C string.

**OZ_int**

```
OZ_Term OZ_int(int i)
```

Converts C integer `i` to an Oz integer.

**OZ_intToC**

```
int OZ_intToC(OZ_Term t)
```

Converts Oz integer `t` to a C integer. If the Oz integer doesn't fit into the C integer, the maximal resp. minimal C integer values are used.

**OZ_parseInt**

```
char *OZ_parseInt(char *s)
```

Parse `s` as an Oz integer. Returns a pointer to the next character after the integer or null if `s` does not start with an integer in Oz syntax (see *"The Oz Notation"*).

**OZ_CStringToInt**

```
OZ_Term OZ_CStringToInt(char *s)
```

Converts C string `s` to an Oz integer. `s` must be a valid integer in Oz syntax (see *"The Oz Notation"*).

**OZ_floatToC**

```
double OZ_floatToC(OZ_Term t)
```

Converts Oz float `t` to a C float.

**OZ_float**

```
OZ_Term OZ_float(double f)
```

Converts C float `f` to an Oz float.

**OZ_boolToC**

```
int OZ_boolToC(OZ_Term t)
```

Returns non-zero iff `t` is equal to `true`.

**OZ_parseFloat**

```
char *OZ_parseFloat(char *s)
```

Parse `s` as an Oz float. Returns a pointer to the next character after the float or null if `s` is not an float in Oz syntax (see *"The Oz Notation"*).

**OZ_CStringToFloat**

```
OZ_Term OZ_CStringToFloat(char *s)
```

Converts C string `s` to an Oz float. `s` must be a valid float in Oz syntax (see *"The Oz Notation"*).

**OZ_CStringToNumber**

```
OZ_Term OZ_CStringToNumber(char *s)
```

Converts C string `s` to an Oz number. `s` must be a valid integer or float in Oz syntax (see *"The Oz Notation"*).

**`OZ_toC`**

```
char *OZ_toC(OZ_Term t, int depth, int width)
```

Converts any Oz term `t` to an C string. This functions doesn't check for cycles. A `depth` of *n* means that trees are printed to a depth limit of *n* only, deeper subtrees are abbreviated by „,. A `width` of *n* means that for lists at most *n* elements and for records at most *n* fields are printed, the unprinted elements and fields are printed by „,.

**`OZ_string`**

```
OZ_Term OZ_string(char *s)
```

Converts C string `s` to an Oz string.

**`OZ_stringToC`**

```
char *OZ_stringToC(OZ_Term t, int *n)
```

Converts Oz string `t` to a C string and returns in `n` the length of string.

**`OZ_virtualStringToC`**

```
char *OZ_virtualStringToC(OZ_Term t, int *n)
```

Converts Oz virtual string `t` to a C string. The returned value is overridden with the next invocation of this function.

## 7.7 Term access and construction

Several functions are available to access and construct terms.

**`OZ_label`**

```
OZ_Term OZ_label(OZ_Term term)
```

Returns the label of `term`.

**`OZ_width`**

```
int OZ_width(OZ_Term term)
```

Returns the width of `term`.

**`OZ_tuple`**

```
OZ_Term OZ_tuple(OZ_Term label, int width)
```

Returns a new tuple with label `label` and width `width`. Note that the values of all subtrees are still undefined. Hence a call to this function should be immediately followed by calls to `OZ_putArg`.

**`OZ_putArg`**

```
int OZ_putArg(OZ_Term tuple,int pos,OZ_Term arg)
```

Destructively sets the subtree of tuple `tuple` at `pos` to `arg`. The tuple arguments are numbered starting from 0. Should be only used for tuples created with `OZ_tuple`.

**`OZ_mkTuple`**

```
OZ_Term OZ_mkTuple(OZ_Term label, int width, ...)
```

Returns a new tuple with label `label` and width `width`. All subtrees from 1 to `width` must be given as arguments after `width`. Example:

```
OZ_mkTuple(OZ_atom("f"),2,OZ_atom("a"),OZ_int(1))
```

creates the tuple `f(a 1)`.

**OZ_mkTupleC**

```
OZ_Term OZ_mkTupleC(char *label, int width, ...)
```

Analogously to `OZ_mkTuple`, but expects the label as a C string.

**OZ_getArg**

```
OZ_Term OZ_getArg(OZ_Term tuple, int pos)
```

Returns the subtree of tuple `tuple` at `pos`. The tuple arguments are numbered starting from 0.

**OZ_nil**

```
OZ_Term OZ_nil()
```

Returns the atom `nil`.

**OZ_cons**

```
OZ_Term OZ_cons(OZ_Term head, OZ_Term tail)
```

Returns a binary tuple with label `'|'`, where the first field is `head`, the second is `tail`.

**OZ_head**

```
OZ_Term OZ_head(OZ_Term t)
```

Returns the first field of `t`. `t` must be a tuple with label `'|'` and width 2.

**OZ_tail**

```
OZ_Term OZ_tail(OZ_Term t)
```

Returns the second field of `t`. `t` must be a tuple with label `'|'` and width 2.

**OZ_length**

```
int OZ_length(OZ_Term t)
```

Compute the length of the Oz list `t`. This function returns `-1`, if `t` is not determined and `-2`, if `t` is not a list.

**OZ_toList**

```
OZ_Term OZ_toList(int n, OZ_Term *t)
```

Creates an Oz list out of an array `t` of `n` values.

**OZ_pair**

```
OZ_Term OZ_pair(int n)
```

Returns a mixfix pair, with `n` subtrees. The subtrees are not initialized and must be defined with `OZ_putArg`.

**OZ_pair2**

```
OZ_Term OZ_pair2(OZ_Term left, OZ_Term right)
```

Returns a mixfix pair, where the first field is `left` and the second is `right`.

**OZ_pairA**

```
OZ_Term OZ_pairA(char *left, OZ_Term right)
```

Macro for creating a mixfix pair of an atom and a value. It is defined as:

```
OZ_pair2(OZ_atom(left),right)
```

**OZ_pairAA**

```
OZ_Term OZ_pairAA(char *left, char *right)
```

Macro for creating a mixfix pair of two atoms. It is defined as:

```
OZ_pair2(OZ_atom(left),OZ_atom(right))
```

**OZ_pairAI**

```
OZ_Term OZ_pairAI(char *left, int right)
```

Macro for creating a mixfix pair of an atom and an integer. It is defined as:

```
OZ_pair2(OZ_atom(left),OZ_int(right))
```

**OZ_pairAS**

```
OZ_Term OZ_pairAS(char *left, char *right)
```

Macro for creating a mixfix pair of an atom and a string. It is defined as:

```
OZ_pair2(OZ_atom(left),OZ_string(right))
```

**OZ_subtree**

```
OZ_Term OZ_subtree(OZ_Term record, OZ_Term feature)
```

Returns the subtree of record `record` at `feature`.}

**OZ_record**

```
OZ_Term OZ_record(OZ_Term label, OZ_Term arity)
```

Creates a new record with label `label` and list of features `arity`. Note that the values of all subtrees are still undefined. Hence a call to this function should be immediately followed by calls to `OZ_putSubtree`.

**OZ_recordInit**

```
OZ_Term OZ_recordInit(OZ_Term lbl,OZ_Term propList)
```

Creates a new record with label `lbl`. The property list `propList` contains all features and their subtree as mixfixed pairs.

**OZ_putSubtree**

```
void OZ_putSubtree(OZ_Term record, OZ_Term feature,     OZ_Term newTerm)
```

Destructively sets the subtree of record `record` at `feature` to `newTerm`. Should be only used for records created with `OZ_record` or `OZ_recordInit`.

**OZ_arityList**

```
OZ_Term OZ_arityList(OZ_Term record)
```

Returns the arity of record `record` as an Oz list.

**OZ_adjoinAt**

```
OZ_Term OZ_adjoinAt      (OZ_Term rec,OZ_Term fea,OZ_Term val)
```

Returns a new record by adjoining subtree `val` at feature `fea` to record `rec`.

**`OZ_newVariable`**

```
OZ_Term OZ_newVariable()
```

Creates a new variable.

**`OZ_newName`**

```
OZ_Term OZ_newName()
```

Creates a new name.}

**`OZ_newChunk`**

```
OZ_Term OZ_newChunk(OZ_Term record)
```

Creates a new chunk.

**`OZ_newCell`**

```
OZ_Term OZ_newCell(OZ_Term t)
```

Creates a new cell with initial content `t`.

**`OZ_newPort`**

```
OZ_Term OZ_newPort(OZ_Term s)
```

Creates a new port with stream `s`.

**`OZ_send`**

```
void OZ_send(OZ_Term p, OZ_Term t)
```

Sends value `t` to port `p`.

**`OZ_onToplevel`**

```
int OZ_onToplevel()
```

Returns nonzero iff called on toplevel, i.e. not within a local space.

## 7.8 Exceptions

**`OZ_raise`**

```
OZ_Return OZ_raise(OZ_Term t)
```

Raises exception `t`.

**`OZ_raiseC`**

```
OZ_Return OZ_raiseC(char *label,int arity,...)
```

Raises an exception. The exception is created from the argument list just the way it is done by `OZ_mkTupleC`.

**`OZ_typeError`**

```
OZ_Return OZ_typeError(int pos, char *type)
```

Raises an exception indicating that the argument at position `pos` is of incorrect type. `type` should be a string describing the expected type.

## 7.9 Unification

**OZ_unify**

```
OZ_Return OZ_unify(OZ_Term t1, OZ_Term t2)
```

Unify `t1` and `t2`. Return `OZ_ENTAILED` on success and `OZ_FAILED` on failure.

**OZ_unifyInt**

```
OZ_Return OZ_unifyInt(OZ_Term t1, int i)
```

This is an abbreviation for `OZ_unify(t1, OZ_int(i))`

**OZ_unifyFloat**

```
OZ_Return OZ_unifyFloat(OZ_Term t1, float f)
```

This is an abbreviation for `OZ_unify(t1, OZ_float(f))`

**OZ_unifyAtom**

```
OZ_Return OZ_unifyAtom(OZ_Term t1, char *s)
```

This is an abbreviation for `OZ_unify(t1, OZ_atom(s))`

**OZ_eq**

```
OZ_Return OZ_eq(OZ_Term t1, OZ_Term t2)
```

Return non-null, if `t1` and `t2` reference the same Oz object in the store.

**OZ_eqAtom**

```
OZ_Return OZ_eqAtom(OZ_Term t1, char *s)
```

This is an abbreviation for `OZ_eq(t1, OZ_atom(s))`

**OZ_eqInt**

```
OZ_Return OZ_eqInt(OZ_Term t1, int i)
```

This is an abbreviation for `OZ_eq(t1, OZ_int(i))`

**OZ_eqFloat**

```
OZ_Return OZ_eqFloat(OZ_Term t1, double d)
```

This is an abbreviation for `OZ_eq(t1, OZ_float(d))`

## 7.10 Threads

**OZ_makeRunnableThread**

```
void OZ_makeRunnableThread(OZ_Term proc, OZ_Term *args, int n)
```

Creates a thread with one task to execute the application of `proc` to arguments `args[0],...,args[n-1`

**OZ_getLowPrio**
**OZ_getMediumPrio**
**OZ_getHighPrio**

```
int OZ_getLowPrio() int OZ_getMediumPrio() int OZ_getHighPrio()
```

Return the appropriate thread priorities.

## 7.11 Printing

**OZ_warning**

```
void OZ_warning(char *format ...)
```

Prints a warning message to the standard error device. Can be used like `printf(3)`.

## 7.12 Miscellaneous

**OZ_false**

```
OZ_term OZ_false()
```

Returns the Oz name for the boolean value `false`.

**OZ_true**

```
OZ_term OZ_true()
```

Returns the Oz name for the boolean value `true`.

**OZ_unit**

```
OZ_term OZ_unit()
```

Returns the Oz name for `unit`.

**OZ_smallIntMin**

```
int OZ_smallIntMin()
```

Returns the minimal small integer.

**OZ_smallIntMax**

```
int OZ_smallIntMax()
```

Returns the maximal small integer.

**OZ_featureCmp**

```
int OZ_featureCmp(OZ_Term t1, OZ_Term t2)
```

Compares the features `t1` and `t2`. Returns zero if they are equal, -1 if `t1` is less than `t2` and 1 if `t2` is less than `t1`.

**OZ_suspendOn**

```
OZ_suspendOn(OZ_Term v)
```

Suspends the executing thread on `v`. `v` must be a variable. When `v` gets bound then the thread gets woken by first reexecuting the enclosing C function from the beginning.

**OZ_suspendOn2**
**OZ_suspendOn3**

```
OZ_suspendOn2(OZ_Term v1, OZ_Term v2) OZ_suspendOn3(OZ_Term v1, OZ_Term v2,
```

Like `OZ_suspendOn` , but suspends the executing thread disjunctively on the argument variables.

## 7.13 Garbage collection

Care must be taken about proper interaction with the Oz garbage collector: it does not notice if you store an `OZ_Term` into a global C variable. Therefore it will free the space on the heap occupied by this term, which leads to memory faults. Oz provides functions to explicitly inform the garbage collector about external references to the heap.

**`OZ_protect`**

```
int OZ_protect(OZ_Term *tp)
```

During garbage collection the term `tp` points to is visited and may be moved. Therefore `tp` must be a *pointer* to a term. The location where `tp` points to is modified by the garbage collector.

**`OZ_unprotect`**

```
int OZ_unprotect(OZ_Term *tp)
```

This is the inverse function to `OZ_protect` informing the garbage collector that the reference to the heap is no longer used.

**`OZ_gCollect`**

```
int OZ_gCollect(OZ_Term *tp)
```

This function causes the Oz term referred to by `tp` to be updated during garbage collection.

**`OZ_sClone`**

```
int OZ_sClone(OZ_Term *tp)
```

This function causes the Oz term referred to by `tp` to be updated during cloning.

## 7.14 Concurrent Input and Output

Reading from or writing to a file descriptor may block, since buffers my be empty or resp. full. Thus calling `read` or `write` might block the whole Oz process. We therefor provide abstractions that allow concurrent access to file descriptors from within the C level.

We first declare an abstact type `OZ_IOHandler` which is a function expecting an integer and an arbitrary pointer:

```
typedef int OZ_IOHandler(int, void *);
```

The user can then use the following abstractions:

**`OZ_registerReadHandler`**

```
void OZ_registerReadHandler(int fd,OZ_IOHandler fun,void *args)
```

Registers `fun` as a read handler for file descriptor `fd`. Any previously registered function will be overridden. When input gets available on `fd` then `fun(fd,args)` will be called by the Oz scheduler. The usage of `args` provides a way to pass arbitrary arguments to `fun`.

**`OZ_unregisterRead`**

`void OZ_unregisterRead(int fd)`

Unregisters a previously registered read handler for file descriptor `fd`.

**`OZ_registerWriteHandler`**

`void OZ_registerWriteHandler(int fd,OZ_IOHandler fun,void *args)`

Analogously to `OZ_registerReadHandler` for writing. `fun` is called as soon as the output buffer for `fd` gets empty.

**`OZ_unregisterWrite`**

`void OZ_unregisterWrite(int fd)`

Unregisters a previously registered write handler for file descriptor `fd`.

8

# The Extension class

The C++ class `OZ_Extension` allows for an easy integration of new built-in data types into the Oz VM.

To add a new data type a native module must be implemented which contains (1) a subclass of `OZ_Extension` (see below) and (2) built-in procedures implementing the operation on the new type.

If you want to implement situated extension, i.e. data types which are situated in computation spaces and need to be copied you should subclass `OZ_SituatedExtension` with has the same interface as `OZ_Extension`.

In Section 8.1 you find the reference documentation and in Section 8.2 an example.

## 8.1 Reference

### 8.1.1 The class `OZ_Extension`

The class `OZ_Extension` implements the methods defined below. The methods `getIdV`, `gCollectV`, and `sCloneV` which are marked as required are pure virtual and have to be implemented in every subclass.

`virtual int getIdV()` // **required**

Allows to discriminate the different kinds of extensions. It should return a unique number. Unique numbers can be obtained using `int OZ_getUniqueId()`.

Ids can be statically assigned by extending the enumeration `OZ_Registered_Extension_Id`.

`virtual OZ_Extension* gCollectV()` // **required**

Needed for garbage collection.

`virtual OZ_Extension* sCloneV()` // **required**

Needed for cloning of computation spaces (for `OZ_SituatedExtension`).

`virtual void gCollectRecurseV()` // **required**

Invoked on the copy obtained from `gCollectV`. The VM has marked the data such that recursive updates of fields, e.g. running `OZ_gCollect`, does not run into cycles.

`virtual void sCloneRecurseV()` // **required**

Invoked on the copy obtained from `sCloneV`. The VM has marked the data such that recursive updates of fields, e.g. running `OZ_sClone`, does not run into cycles.

`virtual OZ_Term printV(int depth = 10)` // **default: return `extension`**

printV should return a virtual string and is used for printing, e.g. System.show.

`virtual OZ_Term printLongV(int depth = 10, int offset = 0)` // **default: call printV**

This may help debugging, but is currently not used.

`virtual OZ_Term typeV()` // **default: return `extension`**

typeV should return an atom describing the type of the extension. This value is also return in Value.status and Value.type. It should not conflict with the built-in types.

`virtual OZ_Term inspectV()` // **default: call typeV**

Not used. Idea: hook for debugging tools to get information.

`virtual Oz_Boolean isChunkV()` // **default: return true**

Define this to return false is the extension in not a subtype of chunk.

`virtual OZ_Term getFeatureV(OZ_Term fea)` // **default: return 0**

If the operator `.` (dot) is applied to an extension this function is called. If `getFeatureV` returns 0 an exception is raised that the feature is not available.

`virtual OZ_Return getFeatureV(OZ_Term,OZ_Term&)` // **default: return `OZ_FAILED`**

The is the more basic version of the above, where a reference to the return value is passed as the 2nd argument.

`virtual OZ_Return putFeatureV(OZ_Term,OZ_Term)` // **default: return `OZ_FAILED`**

Feature is 1st argument, new value is 2nd argument. This is used e.g. for `:=` support.

`virtual OZ_Return eqV(OZ_Term t)` // **default: return false**

This function is called during unification and equality test (==), if both values are extensions. Implement it if you need structural equality. Note that in this case `isChunkV` should return false.

`virtual OZ_Boolean toBePickledV()` // **default: return false**

return true if pickling is defined.

`virtual OZ_Boolean pickleV(MarshalerBuffer *)` // **default: return false**

This is the hook to pickling. It is only called when `toBePickledV` returned true, and must return true by itself. It writes the extension's external representation into the given buffer (see `oz_registerExtension` below).

`virtual OZ_Boolean marshalSuspV(OZ_Term te, ByteBuffer *, GenTraverser *)` // **default: return false**

This is the hook to serialization for Oz distribution. It returns true if the extension has been serialized, and writes its external representation/a fragment of into the given buffer (see `oz_registerExtension` below). It returns false if the extension cannot be serialized.

`virtual int minNeededSpace()` // **default: return 0**

This method is used by the Oz distribution's serialization routine. It must return the minimal possible size of the next fragment of the extension's external representation. This number must be consistent with the `marshalSuspV` method described above.

`OZ_Boolean isLocal()`

Returns true if a situated extensions is local to the current space.

### 8.1.2 Functions

`Oz_Boolean OZ_isExtension(OZ_Term t)`

Tests if the OZ_Term is an extension (`t` must be dereferenced first).

`OZ_Extension * OZ_getExtension(OZ_Term t)`

Unbox the OZ_Term into an extension (`t` must be dereferenced).

`OZ_Term OZ_extension(OZ_Extension *e)`

Box the Extension into an OZ_Term.

```
typedef OZ_Term (*oz_unmarshalProcType)(MarshalerBuffer *)
typedef OZ_Term (*oz_suspUnmarshalProcType)(ByteBuffer*, GTAbstractEntity* &)
typedef OZ_Term (*oz_unmarshalContProcType)(ByteBuffer*, GTAbstractEntity*)
void oz_registerExtension(int id, oz_unmarshalProcType, oz_suspUnmarshalProcType, oz_unmarsha
```

Registers unmarshal procedures for the extension with the given id. The second and the third procedures can be 0 if marshaling for Oz distribution for the given id is not defined.

`int OZ_newUniqueId()`

Returns a new unique number usable as the id of an extension.

## 8.2 Example

The following is a snippet from the implementation of bit arrays (bitarray.cc). Look at e.g. `mozart/contrib/gdbm/gdbm.cc` for an exampl of a non-system extension.

```
#define BITS_PER_INT (sizeof(int) * 8)

class BitArray: public OZ_Extension {
private:
  int lowerBound, upperBound;
  int *array;
  ...
public:
  virtual
  int getIdV() { return OZ_E_BITARRAY; }

  virtual
```

```
  OZ_Term printV(int depth = 10) { return oz_atom("<BitArray>"); }

  virtual
  OZ_Term typeV() { return oz_atom("bitArray"); }

  virtual
  OZ_Term printLongV(int depth = 10, int offset = 0) {
    return
      OZ_mkTupleC("#",4,
                  OZ_atom("bit array: "), OZ_int(upperBound - lowerBound -
                  OZ_atom(" bits at "),   OZ_int((int)this));
  }

  virtual OZ_Return getFeatureV(OZ_Term,OZ_Term&);
  virtual OZ_Return putFeatureV(OZ_Term,OZ_Term );

  virtual OZ_Extension *gCollectV(void);
  virtual OZ_Extension *sCloneV(void);
  virtual void sCloneRecurseV(void) {}
  virtual void gCollectRecurseV(void) {}

  ...
  BitArray(int lower, int upper): OZ_Extension() {
    ...
  }
  BitArray(const BitArray *b): OZ_Extension() {
    ...
  }
  Bool checkBounds(int i) {
    return lowerBound <= i && i <= upperBound;
  }
  ...
  void set(int);
  ...
};

inline
Bool oz_isBitArray(TaggedRef term) {
  return oz_isExtension(term) &&
    tagged2Extension(term)->getIdV() == OZ_E_BITARRAY;
}

inline
BitArray *tagged2BitArray(TaggedRef term) {
  Assert(oz_isBitArray(term));
  return (BitArray *) tagged2Extension(term);
}

OZ_Extension *BitArray::gCollectV(void) {
```

```
  BitArray *ret = new BitArray(this);
  return ret;
}

OZ_Extension *BitArray::sCloneV(void) {
  BitArray *ret = new BitArray(this);
  return ret;
}

void BitArray::set(int i) {
  Assert(checkBounds(i));
  int relative = i - lowerBound;
  array[relative / BITS_PER_INT] |= 1 << (relative % BITS_PER_INT);
}

#define oz_declareBitArray(ARG,VAR)              \
BitArray *VAR;                                   \
{                                                \
  OZ_declareDetTerm(ARG,_VAR);                   \
  if (!OZ_isBitArray(oz_deref(_VAR))) {          \
    return OZ_typeError(ARG,"BitArray");         \
  } else {                                       \
    VAR = tagged2BitArray(OZ_deref(_VAR));       \
  }                                              \
}

OZ_BI_define(BIbitArray_new,2,1)
{
  OZ_declareInt(0,l);
  OZ_declareInt(1,h);
  if (l <= h)
    OZ_RETURN(OZ_extension(new BitArray(l, h)));
  else
    return OZ_raise(E_ERROR,E_KERNEL,"BitArray.new",2,OZ_in(0),OZ_in(1));
} OZ_BI_end

OZ_BI_define(BIbitArray_is,1,1)
{
  OZ_declareDetTerm(0,x);
  OZ_RETURN_BOOL(oz_isBitArray(oz_deref(x)));
} OZ_BI_end

OZ_BI_define(BIbitArray_set,2,0)
{
  OZ_declareBitArray(0,b);
  OZ_declareInt(1,i);
  if (b->checkBounds(i)) {
    b->set(i);
    return PROCEED;
```

```
  } else
    return OZ_raise(E_ERROR,E_KERNEL,"BitArray.index",2,OZ_in(0),OZ_in(1));
} OZ_BI_end

OZ_Return BitArray::getFeatureV(OZ_Term f,OZ_Term& v)
{
  if (!OZ_isInt(f)) { oz_typeError(1,"int"); }
  int i = OZ_intToC(f);
  if (checkBounds(i)) {
    v = test(i)? OZ_true(): OZ_false();
    return PROCEED;
  } else {
    return OZ_raise(E_ERROR,E_KERNEL,"BitArray.index",2,
                    OZ_extension(this),f);
  }
}

OZ_Return BitArray::putFeatureV(OZ_Term f,OZ_Term v)
{
  if (!OZ_isInt(f)) { oz_typeError(1,"int"); }
  int i = OZ_intToC(f);
  if (!checkBounds(i)) {
    return oz_raise(E_ERROR,E_KERNEL,"BitArray.index",2,
                    OZ_extension(this),f);
  }
  if (OZ_isVariable(v)) { OZ_suspendOn(v); }
  v = oz_deref(v);
  if (v==OZ_true()) set(i);
  else if (v==OZ_false()) clear(i);
  else { return OZ_typeError(2,"bool"); }
  return PROCEED;
}
```

# Index