The Mozart Constraint Extensions Reference

Tobias Müller

Abstract

This reference manual explains all abstractions provided to extend Mozart Oz 3 constraint capabilities. It is intended to be used in conjunction with the document *"The Mozart Constraint Extensions Tutorial"*.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

Implementing Propagators

1.1 Overview

This reference is intended to be a supplement to the user manual. It is assumed that the reader has already read the manual and is familiar with the concepts of the CPI .

Include Files   The value of the environment variable OZHOME is supposed to denote the installation directory of the Oz system to be used. The abstractions provided by the CPI are defined in the following header file.

```
$OZHOME/include/mozart_cpi.hh
```

This file includes the file mozart.h which provides the basic functionality for interfacing Oz with C/C++ code. For details see *"Interfacing to C and C++"*.

Naming Conventions   The CPI defines classes, functions, macros etc. Their names begin with OZ_. Names of macros are made up of upper case letters. Member functions and data members begin with lower case letters. The name of accessor functions begin with get and names of predicates begin with is.

The C/C++ compiler to be used is *gcc* version 2.7.2 or higher.

1.2 The class `OZ_Expect`

The functionality provided by class OZ_Expect is intended to be used for implementing header functions.

1.2.1 Types

data type **OZ_expect_t**

```cpp
struct OZ_expect_t {
  int size, accepted;
  OZ_expect_t(int s, int a) : size(s), accepted(a) {}
};
```

Return type of member functions which check for constraints on parameters (see Section 1.2.4).

enumerable type **OZ_FDPropState**

```
enum
OZ_FDPropState {fd_prop_singl = 0,
                fd_prop_bounds,
                fd_prop_any};
```

The values of this enumerable type are used to determine what kind of pruning of a finite domain causes a propagator to be resumed. The values OZ_FDPropState have the following meaning.

value	rerun propagator in case ...
fd_prop_singl	... a finite domain becomes a singleton.
fd_prop_bounds	... the bounds of a finite domain are narrowed.
fd_prop_any	... an arbitrary value is removed from a finite domain or an equality constraint is imposed.

enumerable type **OZ_FSetPropState**

```
enum
OZ_FSetPropState {fs_prop_glb = 0,
                  fs_prop_lub,
                  fs_prop_val,
                  fs_prop_any,
                  fs_prop_bounds};
```

The values of this enumerable type are used to determine what kind of pruning of a finite set constraint causes a propagator to be resumed. The values OZ_FSetPropState have the following meaning.

value	rerun propagator in case ...
fs_prop_val	... a finite set constraint becomes a finite set value.
fs_prop_glb	... a value is added to a set.
fs_prop_lub	... a value is removed from a set.
fs_prop_bounds	... a value is added to or removed from a set.
fs_prop_any	... either the conditions for fs_prop_bounds apply or an equality constraint is imposed.

data type **OZ_ExpectMeth**

```
typedef
OZ_expect_t (OZ_Expect::*OZ_ExpectMeth)(OZ_Term)
```

Type of member functions which check for constraints on parameters (see Section 1.2.4).

1.2.2 Constructor and Destructor

OZ_expect

```
OZ_Expect(void);
```

Default constructor of the class OZ_Expect.

destructor **~OZ_expect**

```
~OZ_Expect();
```

Destructor of the class OZ_Expect.

1.2.3 Specification of a Set of Integers

Specification of a set of integers is mainly used in context with finite domain and finite set constraints.

$$
\begin{array}{rcl}
\textit{level\_4} & ::= & \textit{level\_3} \\
& | & \text{compl}(\textit{level\_3})
\end{array}
$$

$$
\begin{array}{rcl}
\textit{level\_3} & ::= & \textit{level\_2} \\
& | & [\textit{level\_2}+] \\
& | & \texttt{nil}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{level\_2} & ::= & \textit{level\_1} \\
& | & \textit{level\_1}\texttt{\#}\textit{level\_1}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{level\_1} & ::= & \texttt{OZ\_getFDInf(),...,OZ\_getFDSup()} \\
& & (\text{in case of } \texttt{OZ\_Expect::expectDomDescr()}) \\
& | & \texttt{OZ\_getFSetInf(),...,OZ\_getFSetSup()} \\
& & (\text{in case of } \texttt{OZ\_Expect::expectFSetDescr()})
\end{array}
$$

1.2.4 Member Functions for Checking the Constraint Store

A member function described in this section takes as first argument a term, typically a parameter of a propagator. Extra arguments allow to control the behaviour of the member function or to specify the way subterms are to be checked. The returned value is of type `OZ_expect_t` and denotes the result of the examination of the constraint store.

expectDomDescr

```
OZ_expect_t expectDomDescr(OZ_Term descr, int level = 4);
```

This member function expects `descr` to be a finite domain specification (see Section 1.2.3) according to `level`. The non-terminal *level_n* in Section 1.2.3 corresponds to `level=n`.

expectFSetDescr

```
OZ_expect_t expectFSetDescr(OZ_Term descr, int level = 4);
```

This member function expects `descr` to be a finite set specification (see Section 1.2.3) according to `level`. The non-terminal *level_n* in Section 1.2.3 corresponds to `level=n`.

expectVar

```
OZ_expect_t expectVar(OZ_Term t);
```

Expects `t` to be a variable. A determined term `t` is regarded as an inconsistency.

expectRecordVar

```
OZ_expect_t expectRecordVar(OZ_Term t);
```

Expects `t` to be a record.

expectBoolVar

```
OZ_expect_t expectBoolVar(OZ_Term t);
```

Expects t to be a finite domain variable with domain $\{0, 1\}$ resp. the value 0 or 1.

expectIntVar

```
OZ_expect_t
expectIntVar(OZ_Term t,
             OZ_FDPropState ps = fd_prop_any);
```

Expects t to be a finite domain variable or a finite domain integer. The value of ps controls on what events the propagator has to be resumed. See the explanation on `OZ_FDPropState` in Section 1.2.1 for the values of ps.

expectFSetVar

```
OZ_expect_t
expectFSetVar(OZ_Term t,
              OZ_FSetPropState ps = fs_prop_any);
```

Expects t to be a finite set variable or a finite set value. The value of ps controls on what events the propagator has to be resumed. See the explanation on `OZ_FSetPropState` in Section 1.2.1 for the values of ps.

expectGenCtVar

```
OZ_expect_t expectGenCtVar(OZ_Term t,
                           OZ_CtDefinition * def,
                           OZ_CtWakeUp w);
```

Expects t to be a constrained variable resp. a compatible value according to `def`. The value w determines the event the propagator is reinvoked. See Section 2.1 for details on `OZ_CtDefinition` and Section 2.2 for details on `OZ_CtWakeUp`.

expectInt

```
OZ_expect_t expectInt(OZ_Term t);
```

Expects t to be a small integer. See the systems manual *"Interfacing to C and C++"* for details.

expectFloat

```
OZ_expect_t expectFloat(OZ_Term t);
```

Expects t to be a float.

expectFSetValue

```
OZ_expect_t expectFSetValue(OZ_Term t);
```

Expects t to be a finite set value.

expectLiteral

```
OZ_expect_t expectLiteral(OZ_Term t);
```

Expects t to be a literal.

expectLiteralOutOf

```
OZ_expect_t expectLiteralOutOf(OZ_Term t, OZ_Term * ls);
```

Expects t to be a literal contained in `ls` where `ls` points to an array of literals terminated with `(OZ_Term) NULL`.

expectVector

```
OZ_expect_t expectVector(OZ_Term t,
                         OZ_ExpectMeth expect_f);
```

Expects `t` to be a vector of terms which are all sufficiently constrained with respect to `expect_f`. A vector is either a tuple, a closed record, or a list.

expectProperRecord

```
OZ_expect_t expectProperRecord(OZ_Term t,
                               OZ_ExpectMeth expect_f);
```

Expects `t` to be a proper record where all subtrees are sufficiently constrained with respect to `expect_f`. A proper record expects its subtrees to be indexed by literals.

expectProperRecord

```
OZ_expect_t expectProperRecord(OZ_Term t,
                               OZ_Term * ar);
```

Expects `t` to be a proper record with at least subtrees under the features in `ar` are present where `ar` points to an array of features terminated with `(OZ_Term) NULL`.

expectProperTuple

```
OZ_expect_t expectProperTuple(OZ_Term t,
                              OZ_ExpectMeth expect_f);
```

Expects `t` to be a proper tuple where all subtrees are sufficiently constrained with respect to `expect_f`. A proper tuple expects its subtrees to be indexed by integers.

expectList

```
OZ_expect_t expectList(OZ_Term t, OZ_ExpectMeth expect_f);
```

Expects `t` to be a list where all elements are sufficiently constrained with respect to `expect_f`. A list is either the atom `nil` or a 2-tuple with label `'|'` where the second element is a list again.

expectStream

```
OZ_expect_t expectStream(OZ_Term st);
```

Expects either an unbound variable or `nil` resp. a 2-tuple with label `'|'` where the second element is a stream too.

1.2.5 Member Functions for Control Purposes

collectVarsOn

```
void collectVarsOn(void);
```

This member function turns collecting variables *on*. That means that pruning of parameters checked in this mode may cause the propagator to be resumed.

collectVarsOff

```
void collectVarsOff(void);
```

This member function turns collecting variables *off*. That means that pruning of parameters checked in this mode *cannot* cause the propagator to be resumed.

impose

```
OZ_Return impose(OZ_Propagator *p);
```

The propagator `p` is imposed. The return value is the result of the initial invocation of
`OZ_Propagator::propagate()`.

suspend

```
OZ_Return suspend(OZ_Thread th);
```

This member function is to be called if the header function has to be suspended. The
thread `th` can be created with `OZ_makeSuspendedThread()` which is defined by the
SCI (see *"Interfacing to C and C++"* for details).

fail

```
OZ_Return fail(void);
```

This member function is to be called if an inconsistency has been detected.

isSuspending

```
OZ_Boolean isSuspending(OZ_expect_t r);
```

Returns `OZ_TRUE` if `r` indicates that constraints expected on a parameter are not present
in the current store. Otherwise it returns `OZ_FALSE`.

isFailing

```
OZ_Boolean isFailing(OZ_expect_t r);
```

Returns `OZ_TRUE` if `r` indicates an inconsistency. Otherwise it returns `OZ_FALSE`.

isExceptional

```
OZ_Boolean isFailing(OZ_expect_t r);
```

Returns `OZ_TRUE` if `r` indicates an inconsistency causing an exception. Otherwise it
returns `OZ_FALSE`.

1.2.6 Macros

macro **OZ_EXPECTED_TYPE**

```
OZ_EXPECTED_TYPE(S)
```

This macro declares a C/C++ string used by the macros `OZ_EXPECT` and `OZ_EXPECT_SUPEND`
in case an inconsistency is detected. For details see Section *Imposing Nestable Propa-
gators*, *(The Mozart Constraint Extensions Tutorial)*.

macro **OZ_EM**

The macros `OZ_EM_` are provided to create standardized error messages.

expected constraint	macro to be used
literal	`OZ_EM_LIT`
float	`OZ_EM_FLOAT`
small integer	`OZ_EM_INT`
finite domain integer	`OZ_EM_FD`
boolean finite domain integer in $\{0, 1\}$	`OZ_EM_FDBOOL`
description of a finite domain integer	`OZ_EM_FDDESCR`
finite set of integers	`OZ_EM_FSETVAL`
finite set of integers constraint	`OZ_EM_FSET`
description of a finite set of integers	`OZ_EM_FSETDESCR`
vector of	`OZ_EM_VECT`
record of	`OZ_EM_RECORD`
truth name	`OZ_EM_TNAME`
stream	`OZ_EM_STREAM`

macro **`OZ_EXPECT`**

> `OZ_EXPECT(O, P, F)`

This macros checks if a term occurring at argument position `P` of a SCI function is currently expectedly constrained with respect to the function `F`. The first parameter `O` must be an instance of the class `OZ_Expect` resp. a class derived from it. Only if the expected constraints are available in the store the code following this macro is executed.

macro **`OZ_EXPECT_SUSPEND`**

> `OZ_EXPECT_SUSPEND(O, P, F, SC)`

This macros has the same semantics as the previous one except that in case that expected constraints are currently not present in the store the counter `SC` is incremented and the following code is executed.

1.3 The class `OZ_Propagator`

This class is the base class of all propagators to be implemented. Since this class is a virtual base class, it is not possible to create an instance of that class.

1.3.1 Constructor and Destructor Member Functions

`OZ_Propagator`

> `OZ_Propagator(void);`

This constructor is to be called whenever an instance of a class derived from `OZ_Propagator` is created.

`~OZ_Propagator`

> `virtual ~OZ_Propagator();`

This destructor is defined to be virtual to force the destructors in the derived classes to be virtual. This ensure that destroying a derived class results in calling the destructor of the derived class.

1.3.2   Operator Member Functions

new

```
static void * operator new(size_t);
```

This operator allocates the appropriate amount of heap memory when a propagator is created.

delete

```
static void operator delete(void *, size_t);
```

This operator deallocates the heap memory occupied by a propagator when it is destroyed.

1.3.3   Provided Member Functions

mayBeEqualVars

```
OZ_Boolean mayBeEqualVars(void);
```

This member function returns `OZ_TRUE` if at least one variable the propagator was imposed on has been unified. Otherwise it returns `OZ_FALSE`. See Section *Detecting Equal Variables in a Vector*, *(The Mozart Constraint Extensions Tutorial)* for details.

replaceBy

```
OZ_Return replaceBy(OZ_Propagator * p);
```

This member function replaces the current propagator (i.e. `*this`) by the propagator `p`.

replaceBy

```
OZ_Return replaceBy(OZ_Term a, OZ_Term b);
```

This member function replaces the current propagator (i.e. `*this`) by the equality constraint between `a` and `b`.

Caution: before `replaceBy` can be called, for all `x` of type `OZ_FDIntVar` the member function `x.leave()` has to be called.

replaceByInt

```
OZ_Return replaceByInt(OZ_Term v, int i);
```

This member function replaces the current propagator (i.e. `*this`) by the equality constraint between `v` and `i`.

postpone

```
OZ_Return postpone(void);
```

This member function (usually in conjunction with the `return` statement) causes the execution of the propagator to be postponed, i.e. the propagator is immediately switched to `runnable` and put at the end of the thread queue.

imposeOn

```
OZ_Boolean imposeOn(OZ_Term t);
```

This member function imposes the current propagator (i.e. `*this`}) on `t`. If the imposition was successful, i.e., `t` refers to a variable, `OZ_TRUE` is returned, otherwise `OZ_FALSE`.

addImpose

```
void addImpose(OZ_FDPropState s, OZ_Term v);
void addImpose(OZ_FSetPropState s, OZ_Term v);
```

These member functions add `v` to the parameters of the propagator to be imposed with next invocation of `OZ_Propagator::impose`. In case `v` does not denote a variable nothing happens. The value of `s` determines the event when the propagator is to be resumed.

impose

```
void impose(OZ_Propagator * p);
```

This member function imposes the propagator `p` on the parameters collected by `addImpose`. The propagator is immediately switched to *runnable*, but not initially run.

toString

```
char * toString(void) const;
```

Returns a textual representation of the propagator pointing to a static array of `char`s.

1.3.4 Member Functions to be Defined by the Programmer

The member functions in this section are purely virtual, i.e., a class inheriting from `OZ_Propagator` *must* define these functions, otherwise it is not possible to create instances of such a class. These pure virtual member functions make `OZ_Propagator` to an abstract base class.

sizeOf

```
virtual size_t sizeOf(void) = 0;
```

The implementation of this pure virtual function in a derived class `P` is supposed to return the size of an instance of `P`.

sClone

```
virtual void sClone(void) = 0;
```

The implementation of this pure virtual function in a derived class `P` is called during cloning and is supposed to apply to each data member of type `OZ_Term` the function `OZ_sCloneTerm` (see Section 1.9) and possibly, copy dynamically allocated extensions of the object's state. Further details on that issue can be found in Section *Avoiding Redundant Copying*, *(The Mozart Constraint Extensions Tutorial)*.

gCollect

```
virtual void gCollect(void) = 0;
```

The implementation of this pure virtual function in a derived class `P` is called during garbage collection and is supposed to apply to each data member of type `OZ_Term` the function `OZ_sCloneTerm` (see Section 1.9) and possibly, copy dynamically allocated extensions of the object's state. Further details on that issue can be found in Section *Avoiding Redundant Copying*, *(The Mozart Constraint Extensions Tutorial)*.

propagate

```
virtual OZ_Return propagate(void) = 0;
```

The implementation of this pure virtual function in a derived class `P` is supposed to implement the operational semantics of the propagator. The return value indicates the result of the computation to the emulator.

getParameters

```
virtual OZ_Term getParameters(void) const = 0;
```

The implementation of this pure virtual function in a derived class `P` is supposed to return the list (as Oz data structure) of `P`'s parameters. Nested parameter structures are to be represented as nested lists.

getProfile

```
virtual OZ_PropagatorProfile getProfile(void) const = 0;
```

The implementation of this pure virtual function in a derived class `P` is supposed to return the static profile member function used to get information about the state of a propagator class (for instance, the number of total invocations).

1.4   The class `OZ_FDIntVar`

An instance of this class is a mapping of a finite domain integer variable on the heap of the emulator to a C/C++ data structure. The provided functionality allows to directly manipulate the domain (constraint) of the heap variable.

1.4.1   Constructor Member Functions

constructor **OZ_FDIntVar**

```
OZ_FDIntVar(void);
```

This constructor creates an uninitialised instance of the class `OZ_FDIntVar`, which can be initialised later by the member functions `ask()`, `read()`, or `readEncap()`.

constructor **OZ_FDIntVar**

```
OZ_FDIntVar(OZ_Term v);
```

This constructor creates an instance of class `OZ_FDIntVar` and initialises it using `read()`.

1.4.2   Operator Member Functions

new

```
static void * operator new(size_t);
```

This operator allocates memory for a single instance of class `OZ_FDIntVar`. This operator must only be used inside the function `propagate()` of class `OZ_Propagator`. The allocated memory is automatically reclaimed when `propagate()` is left.

new[]

```
static void * operator new[](size_t);
```

This operator allocates memory for an array of instances of `OZ_FDIntVar`. This operator must only be used inside the function `propagate()` of class `OZ_Propagator`. The allocated memory is automatically reclaimed when `propagate()` is left.

delete

```
static void operator delete(void *, size_t);
```

This operator is a dummy since reclaiming memory happens automatically.

delete[]

```
static void operator delete[](void *, size_t);
```

This operator is a dummy since reclaiming memory happens automatically.

operator *

```
OZ_FiniteDomain &operator * (void);
```

This operator returns a finite domain representing the constraint of this variable.

operator ->

```
OZ_FiniteDomain * operator -> (void);
```

This operator returns a pointer to a finite domain representing the constraint of this variable.

1.4.3 Member Functions

isTouched

```
OZ_Boolean isTouched(void) const;
```

This function returns `OZ_TRUE` if at least one element has been removed from the domain and otherwise `OZ_FALSE`.

ask

```
void ask(OZ_Term);
```

This member function initialises an instance of `OZ_FDIntVar` for only reading constraints from the store and it does not require a call of `leave()` or `fail()`.

read

```
int read(OZ_Term);
```

This member function initialises an instance of `OZ_FDIntVar` for constraints to be read from and to be written to the constraint store. It returns the size of the domain. Using this function requires to call either `leave()` or `fail()` when leaving the member function `propagate()` of class `OZ_Propagator`.

readEncap

```
int readEncap(OZ_Term);
```

This member function initialises an instance of `OZ_FDIntVar` for constraints to be read from the constraint store and to perform encapsulated constraint propagation as required by reified constraint propagators. It returns the size of the domain. Using this function requires to call either `leave()` or `fail()` when leaving the member function `OZ_Propagator::propagate()`. For further details see Section *Reified Constraints*, *(The Mozart Constraint Extensions Tutorial)*.

leave

```
OZ_Boolean leave(void);
```

This member function has to be applied to each object of type `OZ_FDIntVar` when leaving the function `propagate()` of class `OZ_Propagator` and *no* inconsistency was detected (except it was initialised with `ask()`). This member function returns `OZ_FALSE` if the domain denotes a singleton. Otherwise it returns `OZ_TRUE`.

fail

```
void fail(void);
```

This member function has to be applied to each object of type `OZ_FDIntVar` when leaving the function `propagate()` of class `OZ_Propagator` and inconsistency *was* detected (except it was initialised with `ask()`).

dropParameter

```
void dropParameter(void);
```

This member function removes the parameter associated with `*this` from the parameter set of the current propagator. This function takes care of multiple occurrences of a single variable as parameter, i.e., a parameter is removed if there is only one occurrence of the corresponding variable in the set of parameter left.

1.5 The class `OZ_FiniteDomain`

Instances of this class represent the domains for finite domain integer variables. A domain may have holes and can range from 0 to `OZ_getFDSup()`, which is currently 134 217 726.

The representation of a finite domain consists of two parts. As long as there are no holes in the domain it suffices to store the lower and upper bound of the domain. Holes are stored in the so-called extension of the domain representation. This extension is either a bit-vector or a list of intervals. The kind of extension used is automatically determined and not visible outside.

The smallest element of a domain d is denoted by $min(d)$ and the largest element by $max(d)$.

1.5.1 Miscellaneous

enumerable type **OZ_FDState**

```
enum OZ_FDState {fd_empty, fd_full, fd_bool, fd_singl};
```

Values of this enumerable type are used when constructing an instance of the class `OZ_FiniteDomain` or in conjunction with operators `==` resp. `!=`.

value	explanation
fd_empty	The domain does not contain any element.
fd_full	The domain contains all elements possible, i.e. $0, \ldots,$ `OZ_getFDSup()`.
fd_bool	The domain contains 0 and 1.
fd_singl	The domain contains a single element.

1.5.2 Constructor Member Functions

`OZ_FiniteDomain`

```
OZ_FiniteDomain(void);
```

This default constructor creates an *uninitialized* instance.

`OZ_FiniteDomain`

```
OZ_FiniteDomain(const OZ_FiniteDomain &d);
```

This copy constructor copies the current domain of `d` to `*this`, so that `d` and `*this` denote the same domain but are independent representations of it.

`OZ_FiniteDomain`

```
OZ_FiniteDomain(OZ_FDState state);
```

This constructor creates an object which represents a domain according to the value of `state`. Valid values for `state` are `fd_empty` and `fd_full`.

`OZ_FiniteDomain`

```
OZ_FiniteDomain(OZ_Term t);
```

This constructor is the composition of the default constructor and the member function `initDescr()`.

`OZ_FiniteDomain`

```
OZ_FiniteDomain(const OZ_FSetValue &fs);
```

This constructor initialises `*this` with the values contained in the finite set `fs`.

1.5.3 Initialisation Member Functions

The return value of all initialisation member functions is the size of the domain they initialised.

`initRange`

```
int OZ_FiniteDomain::initRange(int l, int u);
```

Initialises an instance of class `OZ_FiniteDomain` to the domain $\{l, \ldots, u\}$.

In case `l` $>$ `u`, the domain is set to be empty.

`initSingleton`

```
int OZ_FiniteDomain::initSingleton(int l);
```

Initialises an instance of class `OZ_FiniteDomain` to the domain $\{l\}$.

`initDescr`

```
int OZ_FiniteDomain::initDescr(OZ_Term d);
```

Initialises an instance of class `OZ_FiniteDomain` to a domain according to the domain description `d`. The domain description must be conform with *level4* (see syntax definition of a domain description in Section 1.2.4, entry `expectDomDesc`).

`initFull`

```
int OZ_FiniteDomain::initFull(void);
```

Initialises an instance of class `OZ_FiniteDomain` to the domain $\{0, \ldots, $`OZ_getFDSup()`$\}$.

initEmpty

> `int OZ_FiniteDomain::initEmpty(void);`

Initialises an instance of class `OZ_FiniteDomain` to the empty domain.

initBool

> `int OZ_FiniteDomain::initBool(void);`

Initialises an instance of class `OZ_FiniteDomain` to the domain $\{0,1\}$.

1.5.4   Reflection Member Functions

getMidElem

> `int getMidElem(void) const;`

This member function returns the element in the middle of the domain. For the domain d it is $(max(d) - min(d))\ div\ 2$. If this value happens to be a hole the element closest to it will be returned. In case there are two elements with the same distance to the middle of the domain the smaller one will be taken.

getNextSmallerElem

> `int getNextSmallerElem(int v) const;`

This member function returns the largest element in the domain smaller than v. In case v is the smallest element it returns -1.

getNextLargerElem

> `int getNextLargerElem(int v) const;`

This member function returns the smallest element in the domain larger than v. In case v is the largest element it returns -1.

getLowerIntervalBd

> `int getLowerIntervalBd(int v) const;`

This member function returns the smallest value of the interval v belongs to. In case v does not belong to any interval the function returns -1.

getUpperIntervalBd

> `int getUpperIntervalBd(int v) const;`

This member function returns the largest value of the interval v belongs to. In case v does not belong to any interval the function returns -1.

getSize

> `int getSize(void) const;`

This member function returns the size of the domain, i.e. the number of elements in the domain.

getMinElem

> `int getMinElem(void) const;`

This member function returns the smallest element of the domain.

getMaxElem

> `int getMaxElem(void) const;`

This member function returns the largest element of the domain.

getSingleElem

```
int getSingleElem(void) const;
```

This member function returns the element of a singleton domain. In case the domain is not a singleton domain it returns -1.

1.5.5 Operator Member Functions

operator =

```
const OZ_FiniteDomain &operator = (const OZ_FiniteDomain &fd);
```

This assignment operator copies `fd` to its left hand side, so that both domains are the same but are independent of each other.

operator ==

```
OZ_Boolean operator == (const OZ_FDState s) const;
```

This operator returns `OZ_TRUE` if the domain corresponds to the value of `s`. Otherwise it returns `OZ_FALSE`.

operator ==

```
OZ_Boolean operator == (const int i) const;
```

This operator returns `OZ_TRUE` if the domain contains only `i`. Otherwise it returns `OZ_FALSE`.

operator !=

```
OZ_Boolean operator != (const OZ_FDState s) const;
```

This operator returns `OZ_TRUE` if the domain does *not* correspond to the value of `s`. Otherwise it returns `OZ_FALSE`.

operator !=

```
OZ_Boolean operator != (const int i) const;
```

This operator returns `OZ_TRUE` if the domain does *not* contain `i` or contains more than one element. Otherwise it returns `OZ_FALSE`.

operator &

```
OZ_FiniteDomain operator & (const OZ_FiniteDomain &y) const;
```

This member function returns the intersection of the finite domains represented by `y` and `*this`.

operator |

```
OZ_FiniteDomain operator | (const OZ_FiniteDomain &y) const;
```

This member function returns the union of the finite domains represented by `y` and `*this`.

operator ~

```
OZ_FiniteDomain operator ~ (void) const;
```

This member function returns the negation of the finite domain represented by `*this`. The negation is computed by removing all elements in `*this` from $\{0, \ldots, \texttt{OZ\_getFDSup()}\}$.

operator &=

```
int operator &= (const OZ_FiniteDomain &y);
int operator &= (const int y);
```

This member function computes the intersection of the finite domains represented by y and *this and assigns the result to *this. Further, the size of the updated domain is returned.

operator +=

```
int operator += (const int y);
```

This member function adds the element y to the domain represented by *this and returns the size of the updated domain.

operator -=

```
int operator -= (const int y);
```

This member function removes the element y from the domain represented by *this and returns the size of the updated domain.

operator -=

```
int operator -= (const OZ_FiniteDomain &y);
```

This member function removes all elements contained in the domain represented by y from the domain represented by *this and returns the size of the updated domain.

operator <=

```
int operator <= (const int y);
```

This member function removes all elements being larger than y from the domain represented by *this and returns the size of the updated domain.

operator >=

```
int operator >= (const int y);
```

This member function removes all elements being smaller than y from the domain represented by *this and returns the size of the updated domain.

1.5.6 Auxiliary Member Functions

intersectWithBool

```
int intersectWithBool(void);
```

This member function intersects the current domain with the domain $\{0,1\}$ and produces the following return value.

return value	meaning
-2	The resulting domain is empty.
-1	The resulting domain is $\{0,1\}$
otherwise	The remaining element is returned.

constrainBool

```
int constrainBool(void);
```

This member function intersects the current domain with the domain $\{0,1\}$ and returns the size of resulting domain.

isIn

```
OZ_Boolean isIn(int i) const;
```

This member function returns `OZ_TRUE` if `i` is contained in the domain represented by
`*this`. Otherwise it returns `OZ_FALSE`.

copyExtension

```
void copyExtension(void);
```

This member function replaces the current extension of the domain representation by
a copy of it.

disposeExtension

```
void disposeExtension(void);
```

This member function frees the heap memory occupied by the extension of the domain.

toString

```
char * toString(void) const;
```

Returns a textual representation of the finite domain pointing to a static array of `char`s.

1.6 The class `OZ_FSetVar`

An instance of this class is a mapping of a finite set constraint variable on the heap of
the emulator to a C/C++ data structure. The provided functionality allows to directly
manipulate the domain (constraint) of the heap variable.

1.6.1 Constructor Member Functions

constructor **OZ_FSetVar**

```
OZ_FSetVar(void);
```

This constructor creates an uninitialised instance of the class `OZ_FSetVar`, which can
be initialised later by the member functions `ask()`, `read()`, or `readEncap()`.

OZ_FSetVar

```
OZ_FSetVar(OZ_Term v);
```

This constructor creates an instance of the class `OZ_FSetVar` and initialises it using
`read()`.

1.6.2 Operator Member Functions

new

```
static void * operator new(size_t);
```

This operator allocates memory for a single instance of class `OZ_FSetVar`. This opera-
tor must only be used inside the member function `propagate()` of the class `OZ_Propagator`.
The allocated memory is automatically reclaimed when `propagate()` is left.

new[]

```
static void * operator new[](size_t);
```

This operator allocates memory for an array of instances of `OZ_FSetVar`. This operator
must only be used inside the member function `propagate()` of the class `OZ_Propagator`.
The allocated memory is automatically reclaimed when `propagate()` is left.

delete

> `static void operator delete(void *, size_t);`

This operator is a dummy since reclaiming memory happens automatically.

delete[]

> `static void operator delete[](void *, size_t);`

This operator is a dummy since reclaiming memory happens automatically.

operator *

> `OZ_FSetConstraint &operator * (void);`

This operator returns a finite set constraint representing the constraint of this variable.

operator ->

> `OZ_FSetConstraint * operator -> (void);`

This operator returns a pointer to a finite set constraint representing the constraint of this variable.

1.6.3 Member Functions

isTouched

> `OZ_Boolean isTouched(void) const;`

This function returns `OZ_TRUE` if at least one element has been removed from or added to the set and otherwise `OZ_FALSE`.

ask

> `void ask(OZ_Term);`

This member function initialises an instance of `OZ_FSetVar` for only reading constraints from the store and it does not require a call of `leave()` or `fail()`.

read

> `void read(OZ_Term);`

This member function initialises an instance of `OZ_FSetVar` for constraints to be read from and to be written to the constraint store. Using this function requires to call either `leave()` or `fail()` when leaving the member function `propagate()` of class `OZ_Propagator`.

readEncap

> `void readEncap(OZ_Term);`

This member function initialises an instance of `OZ_FSetVar` for constraints to be read from the constraint store and to perform encapsulated constraint propagation as required by reified constraint propagators. Using this function requires to call either `leave()` or `fail()` when leaving the member function `OZ_Propagator::propagate()`. For further details see Section *Reified Constraints*, *(The Mozart Constraint Extensions Tutorial)*.

leave

> `OZ_Boolean leave(void);`

This member function has to be applied to each object of type `OZ_FSetVar` when leaving the function `propagate()` of class `OZ_Propagator` and *no* inconsistency was detected (except it was initialised with `ask()`). If the set constraint denotes a set value this member function returns `OZ_FALSE` and else it returns `OZ_TRUE`.

fail

```
void fail(void);
```

This member function has to be applied to each object of type `OZ_FSetVar` when leaving the function `propagate()` of class `OZ_Propagator` and inconsistency *was* detected (except it was initialised with `ask()`).

dropParameter

```
void dropParameter(void);
```

This member function removes the parameter associated with `*this` from the parameter set of the current propagator. This function takes care of multiple occurrences of a single variable as parameter, i.e., a parameter is removed if there is only one occurrence of the corresponding variable in the set of parameter left.

1.7 The class `OZ_FSetValue`

1.7.1 Miscellaneous

enumerable type **OZ_FSetState**

```
enum OZ_FSetState {fs_empty, fs_full};
```

Used when constructing a Finite Set or with the operator `==`.

value	meaning
fs_empty	the empty set
fs_full	the set $\{$`OZ_getFSInf()`,...,`OZ_getFSSup()`$\}$

1.7.2 Constructor Member Functions

OZ_FSetValue

```
OZ_FSetValue(void);
```

This constructor creates an *uninitialised* Finite Set Value.

OZ_FSetValue

```
OZ_FSetValue(const OZ_FSetConstraint &fsc);
```

`fsc` must have a determined value (i.e. `fsc.isValue()` must be true). A Finite Set is constructed from this value.

OZ_FSetValue

```
OZ_FSetValue(const OZ_Term t);
```

Constructor using a *level4* list description like for Finite Domains (see Section 1.2.4) to create a Finite Set Value.

OZ_FSetValue

```
OZ_FSetValue(const OZ_FSetState state);
```

Creates a Finite Set Value according to `state` (`fs_empty` or `fs_full`).

OZ_FSetValue

```
OZ_FSetValue(int min_elem, int max_elem);
```

Creates a Finite Set Value {*min_elem*,...,*max_elem*}.

1.7.3 Reflection Member Functions

getMinElem

```
int getMinElem(void) const;
```

Returns the smallest element of the set.

getMaxElem

```
int getMaxElem(void) const;
```

Returns the largest element of the set.

getNextLargerElem

```
int getNextLargerElem(int i) const;
```

Returns the next larger Element after i in the set, or −1 if there is none.

getNextSmallerElem

```
int getNextSmallerElem(int i) const;
```

Returns the next smaller Element before i in the set, or −1 if there is none.

getKnownInList

```
OZ_Term getKnownInList(void) const;
```

Returns a *level4*-List (see Section 1.2.4) containing the elements in the set.

getKnownNotInList

```
OZ_Term getKnownNotInList(void) const;
```

Returns a *level4*-List (see Section 1.2.4) containing the elements in the complementary set.

1.7.4 Operator Member Functions

operator ==

```
OZ_Boolean operator == (const OZ_FSetValue &fs) const;
```

Tests equality on sets.

operator <=

```
OZ_Boolean operator <= (const OZ_FSetValue &fs) const;
```

Return `OZ_True` if `*this` is a subset of `fs`.

operator &

```
OZ_FSetValue operator & (const OZ_FSetValue &fs) const;
```

Returns the intersection of `*this` with `fs`.

operator |

```
OZ_FSetValue operator | (const OZ_FSetValue &fs) const;
```

Returns the union of `*this` with `fs`.

operator -

```
OZ_FSetValue operator - (const OZ_FSetValue &fs) const;
```

Returns the elements in `*this` not in `fs`.

operator &=

```
OZ_FSetValue operator &= (const OZ_FSetValue &fs);
```

`*this` is assigned its intersection with `fs`.

operator |=

```
OZ_FSetValue operator |= (const OZ_FSetValue &);
```

`*this` is assigned its union with `fs`.

operator &=

```
OZ_FSetValue operator &= (const int i);
```

If `i` is in `*this`, this function returns $\{i\}$; otherwise the empty set.

operator +=

```
OZ_FSetValue operator += (const int i);
```

`i` is put into `*this`.

operator -=

```
OZ_FSetValue operator-=(const int);
```

`i` is removed from `*this`, if in.

operator -

```
OZ_FSetValue operator-(void) const;
```

Returns the complement of `*this`.

1.7.5 Auxiliary Member Functions

init

```
void init(const OZ_FSetState state);
```

Initializes a Finite Set Value according to `state` (`fs_empty` or `fs_full`).

isIn

```
OZ_Boolean isIn(int i) const;
```

Returns `OZ_True` if `i` is in `*this`.

isNotIn

```
OZ_Boolean isNotIn(int) const;
```

Returns `OZ_True` if `i` is not in `*this`.

getCard

```
int getCard(void) const;
```

Returns the cardinality of `*this`.

getKnownNotIn

```
int getKnownNotIn(void) const;
```

Returns the cardinality of *this' complement.

copyExtension

```
void copyExtension(void);
```

This member function replaces the current extension of the set value representation by a copy of it.

disposeExtension

```
void disposeExtension(void);
```

This member function frees the heap memory occupied by the extension of the set value.

toString

```
char * toString(void) const;
```

Returns a textual representation of the finite set value pointing to a static array of `char`s.

1.8   The class `OZ_FSetConstraint`

An `OZ_FSetConstraint` defines (among other things) a set of values that are definitely in (the greatest lower bound), a set of values that are definitely out of any set satisfying the Constraint; and a set of values who may or may not be in. These sets will be referred to as `IN`, `OUT`, and `UNKNOWN` sets in the descriptions below.

1.8.1   Constructor Member Functions

OZ_FSetConstraint

```
OZ_FSetConstraint(void);
```

Creates an *uninitialised* `OZ_FSetConstraint` entity.

OZ_FSetConstraint

```
OZ_FSetConstraint(const OZ_FSetValue &fs);
```

Creates a constraint where the `IN` set is `fs`.

OZ_FSetConstraint

```
OZ_FSetConstraint(OZ_FSetState state);
```

Creates a Finite Set Constraint with `IN` set of state `state`, and `OUT` its complement.

value of `state`	constraint
`fs_empty`	the empty set matches
`fs_full`	the set $\{0, \ldots, OZ\_getFSetSup()\}$ matches.

OZ_FSetConstraint

```
OZ_FSetConstraint(const OZ_FSetConstraint &fsc);
```

Copy-constructs a Finite Set Constraint from `fsc`.

1.8.2 Initialization Member Functions

init

```
void init(void);
```

Initializes an empty constraint.

init

```
void init(const OZ_FSetValue &fs);
```

Initializes a constraint that is only matched by `fs`.

init

```
void init(OZ_FSetState);
```

Initializes a Finite Set Constraint with `IN` set of state `state`, and `OUT` its complement.

value of state	constraint
`fs_empty`	the empty set matches
`fs_full`	the set $\{0,\ldots,OZ\_getFSetSup()\}$ matches.

1.8.3 Reflection Member Functions

These all access members of `*this`.

getKnownIn

```
int getKnownIn(void) const;
```

Returns the cardinality of `IN`.

getKnownNotIn

```
int getKnownNotIn(void) const;
```

Returns the cardinality of `OUT`.

getUnknown

```
int getUnknown(void) const;
```

Returns the cardinality of `UNKNOWN`.

getGlbSet

```
OZ_FSetValue getGlbSet(void) const;
```

Returns `IN`.

getLubSet

```
OZ_FSetValue getLubSet(void) const;
```

Returns the set of values that *may* be in sets satisfying the constraint.

getUnknownSet

```
OZ_FSetValue getUnknownSet(void) const;
```

Returns `UNKNOWN`.

getNotInSet

```
OZ_FSetValue getNotInSet(void) const;
```

Returns `OUT`.

getGlbCard

```
int getGlbCard(void) const;
```

Returns the cardinality of IN.

getLubCard

```
int getLubCard(void) const;
```

Returns the cardinality of the set of *all* values that are in *some* a set satisfying the constraint.

getNotInCard

```
int getNotInCard(void) const;
```

Returns the cardinality of OUT.

getUnknownCard

```
int getUnknownCard(void) const;
```

Returns the cardinality of UNKNOWN.

iterators

```
int getGlbMinElem(void) const;
int getLubMinElem(void) const;
int getNotInMinElem(void) const;
int getUnknownMinElem(void) const;
int getGlbMaxElem(void) const;
int getLubMaxElem(void) const;
int getNotInMaxElem(void) const;
int getUnknownMaxElem(void) const;
int getGlbNextSmallerElem(int) const;
int getLubNextSmallerElem(int) const;
int getNotInNextSmallerElem(int) const;
int getUnknownNextSmallerElem(int) const;
int getGlbNextLargerElem(int) const;
int getLubNextLargerElem(int) const;
int getNotInNextLargerElem(int) const;
int getUnknownNextLargerElem(int) const;
```

These functions allow to access and iterate over elements of several sets related to the constraint.

name	function
getMinElem	get the minimal element, -1 if empty
getMaxElem	get the maximal element, -1 if empty
getNextLargerElem(i)	get the next larger element above i, -1 if there is none
getNextSmallerElem(i)	get the next smaller element below i, -1 if there is none

name	referred set
glb	the set of values that are in *all* sets satisfying the constraint
lub	the set of *all* values that are in *some* sets satisfying the constraint
unknown	the set of values that are in *some*, but *not all* sets satisfying the constraint
notIn	the set of values that are in *no* sets satisfying the constraint

getCardMin

         `int getCardMin(void) const;`

    Returns the minimal allowed cardinality.

getCardMax

         `int getCardMax(void) const;`

    Returns the maximal allowed cardinality (-1 means the constraint cannot be satisfied)

getCardSize

         `int getCardSize(void) const;`

    Returns the size of the interval between the minimal and maximal allowed cardinality.

getKnownInList

         `OZ_Term getKnownInList(void) const;`

    Returns IN as a list.

getKnownNotInList

         `OZ_Term getKnownNotInList(void) const;`

    Returns OUT as a list.

getUnknownList

         `OZ_Term getUnknownList(void) const;`

    Returns UNKNOWN as a list.

getLubList

         `OZ_Term getLubList(void) const;`

    Returns the union of IN and UNKNOWN as a list.

getCardTuple

         `OZ_Term getCardTuple(void) const;`

    Returns a tuple consisting of integers giving the minimum and maximum allowed cardinality.

1.8.4   Imposing Constraints

    Where an operator member Function returns an `OZ_Boolean`, it is to indicate whether constraint becomes unsatisfiable in the operation.

operator =

         `OZ_FSetConstraint &operator = (const OZ_FSetConstraint &fsc);`

    `fsc` gets assigned to `*this`.

operator -

         `OZ_FSetConstraint operator - (void) const;`

    The complementary constraint is returned.

operator +=

         `OZ_Boolean operator+=(int i);`

`i` is added to `*`**`this`**`.IN`.

operator -=

```
OZ_Boolean operator-=(int i);
```

`i` is added to `*`**`this`**`.OUT`.

operator <<=

```
OZ_Boolean operator <<= (const OZ_FSetConstraint &fsc);
```

`fsc` is added to `*`**`this`**.

operator %

```
OZ_Boolean operator % (const OZ_FSetConstraint &fsc);
```

Returns `OZ_True` if all values known to be in `*`**`this`** are known not to be in `fsc`, and the other way round.

operator &

```
OZ_FSetConstraint operator & (const OZ_FSetConstraint &fsc) const;
```

Returns the intersection of `*`**`this`** and `fsc`.

operator |

```
OZ_FSetConstraint operator | (const OZ_FSetConstraint &fsc) const;
```

Returns the union of `*`**`this`** and `fsc`.

operator -

```
OZ_FSetConstraint operator - (const OZ_FSetConstraint &fsc) const;
```

Returns the difference of `*`**`this`** and `fsc`.

operator <=

```
OZ_Boolean operator <= (const OZ_FSetConstraint &fsc);
```

Returns `OZ_True` if `*`**`this`** has as least the elements excluded (in `OUT`) that are excluded by `fsc`.

operator >=

```
OZ_Boolean operator >= (const OZ_FSetConstraint &);
```

Returns `OZ_True` if `*`**`this`** has as least the elements included (in `IN`) that are included by `fsc`.

operator !=

```
OZ_Boolean operator != (const OZ_FSetConstraint &fsc);
```

The elements known to be in `fsc` are excluded from `*`**`this`**

operator ==

```
OZ_Boolean operator == (const OZ_FSetConstraint &fs) const;
```

Returns `OZ_True` if `*`**`this`** is equivalent to `fsc`.

le

```
OZ_Boolean le(const int i);
```

All values above `i` are excluded from `*`**`this`**.

ge

```
OZ_Boolean ge(const int);
```

All values below `i` are excluded from `*`**`this`**.

1.8.5 Auxiliary Member Functions

putCard

`OZ_Boolean putCard(int cardmin, int cardmax);`

The minimum and maximum allowed cardinality is set.

isValue

`OZ_Boolean isValue(void) const;`

Returns `OZ_True` if the constraint determines exactly one set.

isIn

`OZ_Boolean isIn(int i) const;`

Returns `OZ_True` if `i` is known to be in *every*(!) set satisfying the constraint.

isNotIn

`OZ_Boolean isNotIn(int i) const;`

Returns `OZ_True` if `i` is in *no* set satisfying the constraint.

isEmpty

`OZ_Boolean isEmpty(void) const;`

Returns `OZ_True` if `*this` is satisfied only by the empty set.

isFull

`OZ_Boolean isFull(void) const;`

Returns `true` if `*this` can only be satisfied by the set containing all possible values (i.e. , $\{0, \ldots, OZ\_getFSetSup()\}$).

isSubsumedBy

`OZ_Boolean isSubsumedBy(const OZ_FSetConstraint &fsc) const;`

Returns `true` if `*this` is subsumed by `fsc`.

copyExtension

`void copyExtension(void);`

This member function replaces the current extension of the set constraint representation by a copy of it.

disposeExtension

`void disposeExtension(void);`

This member function frees the heap memory occupied by the extension of the set constraint.

toString

`char * toString(void) const;`

Returns a textual representation of the finite set constraint pointing to a static array of `char`s.

1.9   Auxiliary Interface Functions

function   **OZ_gCollectTerm**

>   void OZ_gCollectTerm(OZ_Term &t);

> During garbage collection this function updates the reference `t` to a term on the heap. This is typically required when the member function `gCollect()` of a propagator is invoked.

function   **OZ_gCollectBlock**

>   void OZ_gCollectBlock(OZ_Term * frm, OZ_Term * to, **const** int n);

> During garbage collection this function updates the `n` elememts in `frm` and stores them in `to`.

function   **OZ_gCollectAllocBlock**

>   OZ_Term * OZ_gCollectAllocBlock(int n, OZ_Term * frm);

> During garbage collection this function updates the `n` elements in `frm` and returns a pointer to the updates. The updates are stored in freshly allocated heap memory.

function   **OZ_sCloneTerm**

>   void OZ_sCloneTerm(OZ_Term &t);

> During cloning this function updates the reference `t` to a term on the heap. This is typically required when the member function `sClone()` of a propagator is invoked.

function   **OZ_sCloneBlock**

>   void OZ_sCloneBlock(OZ_Term * frm, OZ_Term * to, **const** int n);

> During cloning this function updates the `n` elememts in `frm` and stores them in `to`.

function   **OZ_sCloneAllocBlock**

>   OZ_Term * OZ_sCloneAllocBlock(int n, OZ_Term * frm);

> During cloning this function updates the `n` elements in `frm` and returns a pointer to the updates. The updates are stored in freshly allocated heap memory.

function   **OZ_isPosSmallInt**

>   OZ_Boolean OZ_isPosSmallInt(OZ_Term val);

> This function returns `OZ_TRUE` if `val` denotes an integer contained in the finite set $\{0,\ldots,\texttt{OZ\_getFDSup()}\}$. Otherwise it returns `OZ_FALSE`.

function   **OZ_hallocOzTerms**

>   OZ_Term * OZ_hallocOzTerms(int n);

> This function allocates a block of heap memory for `n` items of type `OZ_Term` and returns a pointer to the block.

function   **OZ_hallocChars**

>   char * OZ_hallocChars(int n);

> This function allocates a block of heap memory for `n` items of type `char` and returns a pointer to the block.

function   **OZ_hallocCInts**

>   int * OZ_hallocCInts(int n);

This function allocates a block of heap memory for `n` items of type `int` and returns a pointer to the block.

function **OZ_hfreeOzTerms**

> void OZ_hfreeOzTerms(OZ_Term * ts, int n);

The function frees the heap memory allocated by `OZ_hallocOzTerms()`. The first argument `ts` points to a memory block and the value of `n` must denote the correct size of the block.

function **OZ_hfreeCInts**

> void OZ_hfreeCInts(int * is, int n);

The function frees the heap memory allocated by `OZ_hallocCInts`. The first argument `is` points to a memory block and the value of `n` must denote the correct size of the block.

function **OZ_hfreeChars**

> void OZ_hfreeChars(char * is, int n);

The function frees the heap memory allocated by `OZ_hallocChars()`. The first argument `is` points to a memory block and the value of `n` must denote the correct size of the block.

function **OZ_copyCInts**

> int * OZ_copyCInts(int n, int * is);

Copies `n` `int`s from `is` and returns the location of the copy. If `n` is equal to 0 it returns `(int *) NULL`.

function **OZ_copyChars**

> char * OZ_copyChars(int n, char * cs);

Copies `n` `char`s from `cs` and returns the location of the copy. If `n` is equal to 0 it returns `(char *) NULL`.

function **OZ_findEqualVars**

> int * OZ_findEqualVars(int sz, OZ_Term * ts);

The function expects `ts` to be a pointer to an `OZ_Term` array of size `sz`. It returns an array of `int`s indicating variables which are equal. Suppose that the ith field of the returned array holds one of the following values.

value	explanation
-1	The term stored at that position is not a variable.
i	This is the first occurrence of a variable stored in the array at position i.
$j \neq i$	This is a repeated occurrence of a variable stored at position j in the array. The first occurrence can be found at position j.

The returned `int` array is statically allocated, i.e. it is overridden on every invocation. For details see Section *Detecting Equal Variables in a Vector*, *(The Mozart Constraint Extensions Tutorial)*.

function **OZ_isEqualVars**

> OZ_Boolean OZ_isEqualVars(OZ_Term v1, OZ_Term v2);

This function returns `OZ_TRUE` if `v1` and `v2` refer to the same variable. Otherwise it returns `OZ_FALSE`.

function **OZ_findSingletons**

```
int * OZ_findSingletons(int sz, OZ_Term * ts);
```

The function expects `ts` to be a pointer to an `OZ_Term` array of size `sz` which expects its elements to refer to finite domain variables. It returns an array of `int`s indicating variables which are singletons. Suppose that the *i*th field of the returned array holds one of the following values.

value	explanation
≥ 0	The term stored at that position is a singleton.
otherwise	The term stored at that position is still a finite domain variable.

The returned `int` array is statically allocated, i.e. it is overridden on every invocation.

function **OZ_typeErrorCPI**

```
OZ_Return OZ_typeErrorCPI(char * __typeString,
                          int pos,
                          char * comment);
```

The return value of this function indicates the runtime system that an exception has to be raised. The message printed is composed using the `pos`th substring of `__typeString` and `comment`.

function **OZ_getFDInf**

```
int OZ_getFDInf(void);
```

This function returns the value of the smallest element a finite domain which is represented by an instance of the class `OZ_FiniteDomain` can take. The value returned is 0.

function **OZ_getFDSup**

```
int OZ_getFDSup(void);
```

This function returns the value of the largest element a finite domain which is represented by an instance of the class `OZ_FiniteDomain` can take. The value returned is 134 217 726.

function **OZ_getFSetInf**

```
int OZ_getFSetInf(void);
```

This function returns the value of the smallest element a finite set value which is represented by an instance of the class `OZ_FSetValue` can take. The value returned is 0.

function **OZ_getFSetSup**

```
int OZ_getFSetSup(void);
```

This function returns the value of the largest element a finite set value which is represented by an instance of the class `OZ_FSetValue` can take. The value returned is 134 217 726.

function **OZ_fsetValue**

```
OZ_Term OZ_fsetValue(OZ_FSetValue * fsv);
```

This function converts the finite set value `fsv` to the corresponding `OZ_Term`.

function **OZ_fsetValueToC**

```
OZ_FSetValue * OZ_fsetValueToC(OZ_Term fsv);
```

This function converts `fsv`, referring to a finite set value, to a pointer to the finite set value.

function **OZ_vectorSize**

```
int OZ_vectorSize(OZ_Term t);
```

This function returns the size of a vector. In case `t` is no vector it returns -1.

type	returned value
Literal	The value returned is 0.
List	he value returned is the length of the list.
Tuple	The value returned is the arity of the tuple.
Record	The value returned is the number of features of the record.

function **OZ_getOzTermVector**

```
OZ_Term * OZ_getOzTermVector(OZ_Term t, OZ_Term * v);
```

This function expects `t` to be a vector and `v` to be an array with minimal `OZ_vectorSize`(t) elements. It converts `t` to an `OZ_Term` array and returns a pointer to the next free position in the array `v` after converting `t`. In case `t` is no vector the function returns `NULL`.

function **OZ_getCIntVector**

```
int * OZ_getCIntVector(OZ_Term t, int * v);
```

This function expects `t` to be a vector of small integers and `v` to be an array with minimal `OZ_vectorSize`(t) elements. It converts `t` to an `int` array and returns a pointer to the next free position in the array `v` after converting `t`. In case `t` is no vector the function returns `NULL`.

Building Constraint Systems from Scratch

2.1 The class `OZ_CtDefinition`

getKind

```
virtual int getKind(void) = 0;
```

Returns an integer identifying a constraint system. The integer value has to be unique for each constraint system. Call `int OZ_getUniqueId(void)` to obtain a unique identifier.

getNoOfWakeUpLists

```
virtual int getNoOfWakeUpLists(void) = 0;
```

Returns the numbers of wake-up lists of variables constrained with this kind of constraint. This number corresponds to the number of events which can cause a propagator being imposed on that kind of variable being rerun.

getNamesOfWakeUpLists

```
virtual char ** getNamesOfWakeUpLists(void) = 0;
```

Returns an array (with `getNoOfWakeUpLists()` entries) of strings describing the event(s) associated to the corresponding wake-up list.

getName

```
virtual char * getName(void) = 0;
```

Returns the name of the constraint system. Is used when outputting variables of that kind.

leastConstraint

```
virtual OZ_Ct * leastConstraint(void) = 0;
```

Returns the constraint which is subsumed by or equal to all other constraints of a certain constraint system.

isValidValue

```
virtual OZ_Boolean isValidValue(OZ_Term t) = 0;
```

Returns `OZ_True` if the Oz value referred to by `t` is a value which is in the domain of the constraint system. Otherwise it returns `OZ_False`.

2.2 The class `OZ_CtWakeUp`

init

```
void init(void);
```

Initializes an instance of this class. [1]

isEmpty

```
OZ_Boolean isEmpty(void);
```

Returns `OZ_True` if no wake-up list has to be scanned.

setWakeUp

```
OZ_Boolean setWakeUp(int i);
```

Sets the wake-up list indexed by `i` ($i = 0, \ldots,$ `getNoOfWakeUpLists()`-1) to be scanned.

isWakeUp

```
OZ_Boolean isWakeUp(int i);
```

Returns `OZ_True` if the corresponding wake-up list indexed by `i` is to be scanned.

getWakeUpAll

```
static OZ_CtWakeUp getWakeUpAll(void);
```

Sets all possible wake-up events.

2.3 The class `OZ_CtProfile`

constructor **OZ_CtProfile**

```
OZ_CtProfile(void);
```

Initializes an instance of this class.

init

```
virtual void init(OZ_Ct * c) = 0;
```

Stores a profile according to the constraint referred to by `c`.

2.4 The class `OZ_Ct`

An instance of this class represents a constraint of a certain constraint system.

constructor **OZ_Ct**

```
OZ_Ct(void);
```

Initializes an instance of this class.

isValue()

```
virtual OZ_Boolean isValue(void) = 0;
```

Returns `OZ_True` if the constraint denotes exactly one value of the domain of the constraint system.

---

[1]Note that there is no default constructor for some implementational reasons.

toValue

```
virtual OZ_Term toValue(void) = 0;
```

Returns an Oz value of the value denoted by the constraint. Returned value is only defined if `isValue` yields `OZ_True`.

isValid

```
virtual OZ_Boolean isValid(void) = 0;
```

Returns `OZ_True` if the constraint denotes at least one element of the domain of the constraint system. Otherwise it returns `OZ_False`.

isWeakerThan

```
virtual OZ_Boolean isWeakerThan(OZ_Ct * c) = 0;
```

Returns `OZ_True` if the constraint represented by `*c` subsumes the constraint represented by `*this` instance.

unify

```
virtual OZ_Ct * unify(OZ_Ct * c) = 0;
```

Returns a constraint that approximates all elements of the constraint domain denoted by the constraints `*c` *and* `*this`.

unify

```
virtual OZ_Boolean unify(OZ_Term t) = 0;
```

Returns `OZ_True` if the value denoted by `t` is included in the values approximated by the constraint.

sizeOf

```
virtual size_t sizeOf(void) = 0;
```

Returns the size of an instance of the class derived `OZ_Ct` (analogue to C's **sizeof** operator).

getProfile

```
virtual OZ_CtProfile * getProfile(void) = 0;
```

Returns a constraint profile (see Section 2.3) according to the constraint.

getWakeUpDescriptor

```
virtual OZ_CtWakeUp getWakeUpDescriptor(OZ_CtProfile * p) = 0;
```

Returns a descriptor for the wake-up lists to be scanned (see Section 2.2). This descriptor is computed by comparing the constraint with the profile `p`. Note the profile is usually taken from the constraint before modifying it.

toString

```
virtual char * toString(int) = 0;
```

Returns a textual representation of the constraint.

copy

```
virtual OZ_Ct * copy(void) = 0;
```

Returns a pointer to a copy of the constraint. The memory for the copy is to be allocated by the operator `OZ_Ct::`**new**.

operator new

```
static void * operator new(size_t, int align = sizeof(void *));
```

Allocates memory for an instance of the constraint on the heap of the Oz runtime system.

operator delete

```
static void operator delete(void *, size_t);
```

Deallocates memory of an instance of the constraint from the heap of the Oz runtime system.

2.5   The class `OZ_CtVar`

The constraint system dependent part of a class derived from `OZ_CtVar` stores typically

- a constraint `C`, i.e., an instance of the class representing a constraint,
- a constraint `EC`, i.e., an instance of the class representing a constraint,
- a reference to a constraint `CR`, and
- a constraint profile `CP`.

The constraint `C` is used to handle constraints of global variables. The constraint `EC` is used to handle encapsulate propagation typically occurring in reified constraints. The reference to a constraint `CR` is used to access the actual constraint and thus to be able to modify it. It either points to `C`, `EC`, or directly to the constraint associated with a constrained variable.

2.5.1   Members to be Defined

ctSetValue

```
virtual void ctSetValue(OZ_Term t) = 0;
```

Initializes `C` to the value denoted by `t` and makes `CR` pointing to `C`.

ctRefConstraint

```
virtual OZ_Ct * ctRefConstraint(OZ_Ct * c) = 0;
```

Sets `CR` to `c` and returns `CR`.

ctSaveConstraint

```
virtual OZ_Ct * ctSaveConstraint(OZ_Ct * c) = 0;
```

Stores `c` in `C` and returns a reference to `C`.

ctSaveEncapConstraint

```
virtual OZ_Ct * ctSaveEncapConstraint(OZ_Ct * c) = 0;
```

Stores `c` in `EC` and returns a reference to `EC`.

ctRestoreConstraint

```
virtual void ctRestoreConstraint(void) = 0;
```

Stores `C` at `*CR`.

ctSetConstraintProfile

> **virtual** void ctSetConstraintProfile(void) = 0;

Initializes CP with the profile of CR.

ctGetConstraintProfile

> **virtual** OZ_CtProfile * ctGetConstraintProfile(void) = 0;

Returns CP.

ctGetConstraint

> **virtual** OZ_Ct * ctGetConstraint(void) = 0;

Returns CR.

isTouched

> **virtual** OZ_Boolean isTouched(void) **const** = 0;

Returns OZ_True if current constraint is not implied anymore by the constraint that was present upon calling read() or readEncap().

2.5.2 Provided Members

constructor **OZ_CtVar**

> OZ_CtVar(void);

Initializes an instance of this class.

operator new

> **static** void * **operator new**(size_t);

Allocates memory for an instance of a class derived from OZ_CtVar on the propagator heap of the Oz runtime system.

operator delete

> **static** void **operator delete**(void *, size_t);

Deallocates memory of an instance of a class derived from OZ_CtVar from the propagator heap of the Oz runtime system.

operator new[]

> **static** void * **operator new**[](size_t);

Allocates memory for an array of instances of a class derived from OZ_CtVar on the propagator heap of the Oz runtime system.

operator delete[]

> **static** void **operator delete**[](void *, size_t);

Deallocates memory of an array of instances of a class derived from OZ_CtVar from the propagator heap of the Oz runtime system.

ask

> void ask(OZ_Term);

Initializes an instance of a derived class of OZ_CtVar for reading the constraint of the corresponding variable. The members leave() and fail() *must not* be called.

read

```
void read(OZ_Term);
```

Initializes an instance of a derived class of `OZ_CtVar` for accessing the corresponding variable in the constraint store for constraint propagation. Modifying the constraint is visible in the store. The members `leave()` and `fail()` *must be* called.

readEncap

```
void readEncap(OZ_Term);
```

Initializes an instance of a derived class of `OZ_CtVar` for accessing the corresponding variable in the constraint store for encapsulated constraint propagation (typically used for reified constraints). Modifying the constraint is *not* visible in the store. The members `leave()` and `fail()` *must be* called.

leave

```
OZ_Boolean leave(void);
```

This member function has to be called if the instance of a derived class of `OZ_CtVar` has been initialized by `read()` resp. `readEncap()` and the constraint represented by the propagator is *consistent* with the constraint store. It returns `OZ_False` if the corresponding variable denotes a value. Otherwise it returns `OZ_True`. Further, this member function causes suspending computation to be woken up.

fail

```
void fail(void);
```

This member function has to be called if the instance of a derived class of `OZ_CtVar` has been initialized by `read()`resp. `readEncap()` and the constraint represented by the propagator is *inconsistent* with the constraint store.

dropParameter

```
void dropParameter(void);
```

This member function removes the parameter associated with *`this` from the parameter set of the current propagator. This function takes care of multiple occurrences of a single variable as parameter, i.e., a parameter is removed if there is only one occurrence of the corresponding variable in the set of parameter left.

Employing Linear Programming Solvers

3.1 The Module `LP`

The module `LP` is provided as contribution (being part of the Mozart Oz 3 distribution[1])
and can be accessed either by

```
declare [LP] = {Module.link ['x-oz://contrib/LP']}
```

or by

```
import RI at 'x-oz://contrib/LP'
```

as part of a functor definition.

`{LP.solve` $RIs +ObjFn +Constrs ?OptSol ?RetVal}

Invoke the LP solver. Use `LP.config` for configuring the solver.

$$
\begin{array}{rcl}
\textit{VECTOR\_OF(X)} & ::= & \text{tuple of } X \\
 & | & \text{record of } X \\
 & | & \text{list of } X
\end{array}
$$

$$
\textit{RIs} \quad ::= \quad \textit{VECTOR\_OF(RI)}
$$

$$
\textit{RI} \quad ::= \quad \text{float} \mid \text{real interval variable}
$$

The first parameter is a vector of real-interval variables. The current bounds of the
real-intervals are used as bound constraints by the LP solver. The second parameter
determines the objective function:

$$
\textit{ObjFn} \quad ::= \quad \texttt{objfn(row:} \; \langle \textsf{VECTOR\_OF(float)} \rangle \\
\texttt{opt: min} \; | \; \texttt{max)}
$$

The value at `opt` stands for minimize (`min`) resp. maximize (`max`). The third parameter
introduces the constraints to the LP solver.

---

[1]The module `LP` is *not* provided on any Windows platform.

$$Constrs \quad ::= \quad VECTOR\_OF(Constr)$$

$$Constr \quad ::= \quad \texttt{constr(row:} \quad \langle \mathsf{VECTOR\_OF(float)} \rangle$$
$$\texttt{type:} \ \texttt{'=<'} \mid \texttt{'=='} \mid \texttt{'>='}$$
$$\texttt{rhs:} \quad \texttt{float })$$

The fourth parameter `OptSol` is constrained to the optimal solution. In case it is already constrained to a real-interval variable, the LP solver derives an additional constraint which makes sure that no greater (minimize) resp. smaller (maximize) solution is found. The last parameter indicates the success of the LP solvers.

$$RetVal \quad ::= \quad \texttt{optimal}$$
$$\mid \quad \texttt{infeasable}$$
$$\mid \quad \texttt{unbounded}$$
$$\mid \quad \texttt{failure}$$

{LP.config +put +ConfigDirection**}**

Set configuration of module `LP`. One can set `mode` and `solver`.

{LP.config +get ?CurrentConfig**}**

Read current configuration of module `LP`.

$$CurrentConfig \quad ::= \quad \texttt{config(avail:} \quad \langle \mathsf{AVAIL\_SOLVERS} \rangle$$
$$\mid \qquad \texttt{mode:} \quad \langle \mathsf{MODES} \rangle$$
$$\mid \qquad \texttt{solver:} \ \langle \mathsf{SOLVER} \rangle)$$

Note that $\langle \mathsf{SOLVER} \rangle$ takes a value out of $\langle \mathsf{AVAIL\_SOLVERS} \rangle$. The solvers available depend on your local installation. The solver *LP_SOLVE* (`lpsolve`) is the default solver.

$$AVAIL\_SOLVERS \quad ::= \quad \texttt{lpsolve}$$
$$\mid \quad \texttt{cplex\_primopt}$$
$$\mid \quad \texttt{cplex\_dualopt}$$

The solver may run in two modes:

$$MODES \quad ::= \quad \texttt{quiet}$$
$$\mid \quad \texttt{verbose}$$

The `verbose` mode is intended for debugging and outputs whether an optimal was found (resp. if not what was the problem) and if so the optimal solution.

Propagation Engine Library

4.1 Overview

enumerable type **pf_return_t**

```
typedef enum { pf_failed,
               pf_entailed,
               pf_sleep } pf_return_t;
```

Return type of a propagation function.

function type **pf_fnct_t**

```
typedef pf_return_t (* pf_fnct_t)(int *, PEL_SuspVar * []);
```

Type of a propagation function. A propagation function takes an array of parameter indicies and an array of references to constrained variables (`(PEL_SuspVar *)`). It returns a value of type `pf_return_t`.

4.2 The class `PEL_ParamTable`

Sadd

```
int add(int i);
```

Add parameter index `i` to parameter table. The table index where `i` is stored is returned.

getHigh

```
int getHigh(void);
```

Returns the highest table index of the table.

operator []

```
int &operator [] (int i);
```

Returns a reference to the element at table position `i`, i.e., the element can be read and written.

4.3 The class `PEL_EventList`

add

```
int add(int i);
```

Add propagation function index `i` to event list. The event list index where `i` is stored is returned.

wakeup

```
void wakeup(PEL_PropQueue *pq, PEL_PropFnctTable * pft[]);
```

Copies all entries of the event list to `pq` and marks the appropriate entries in `pft` as *scheduled*.

getHigh

```
int getHigh(void);
```

Returns the highest index of the event list.

operator []

```
int &operator [] (int i);
```

Returns a reference to the element at event list position `i`, i.e., the element can be read and written.

4.4 The class `PEL_PropFnctTableEntry`

constructor **PEL_PropFnctTableEntry**

```
PEL_PropFnctTableEntry(pf_fnct_t fn, int idx);
```

Constructs a propagator table entry with propagation function `fn` and index to parameter table `idx`.

isScheduled

```
void isScheduled(void);
```

Tests if the propagation function of this entry is marked as *scheduled*.

setScheduled

```
void setScheduled(void);
```

Marks the propagation function of this entry as *scheduled*.

unsetScheduled

```
void unsetScheduled(void);
```

Marks the propagation function of this entry as not *scheduled*.

isDead

```
int isDead(void);
```

Tests if the propagation function of this entry is marked as *dead*.

setDead

```
void setDead(void);
```

Marks the propagation function of this entry as *dead*.

getFnct

```
pf_fnct_t getFnct(void);
```

Returns the pointer to the propagation function of this entry.

getParamIdx

```
int getParamIdx(void);
```

Returns the index to parameter table of this entry.

4.5 The class `PEL_PropFnctTable`

constructor **PEL_PropFnctTable**

```
PEL_PropFnctTable(void);
```

Constructs a propagation function table.

add

```
int add(PEL_ParamTable &pt, PEL_PropQueue &pq,
        pf_fnct_t fnct, int x, int y);

int add(PEL_ParamTable &pt, PEL_PropQueue &pq,
        pf_fnct_t fnct, int x, int y, int z);
```

Adds an entry for the propagation function `fnct` with parameters `x` and `y` (resp. `x`, `y`, and `z`) and returns the index of the entry in the table. The propagation function is registered with `pq` and the parameter indices are stored in `pt`.

4.6 The class `PEL_PropQueue`

constructor **PEL_PropQueue**

```
PEL_PropQueue(void);
```

Constructs a propagation queue.

enqueue

```
void enqueue(int fnct_idx);
```

Enqueue a propagation function index `fnct_idx`. The propagation function index is related to a propagation function table.

dequeue

```
int dequeue(void);
```

Returns a propagation function index.

apply

```
pf_return_t apply(PEL_PropFnctTable &pft,
                  PEL_ParamTable &pt,
                  PEL_SuspVar * []);
```

Dequeues an index and applies the corresponding propagation function closure of `pft`. It returns the value returned by the propagation function.

isEmpty

```
int isEmpty(void);
```

Tests if the queue is empty.

setFailed

```
void setFailed(void);
```

Sets the queue *failed*.

isFailed

```
int isFailed(void);
```

Tests if the queue is *failed*.

isBasic

```
int isBasic(void);
```

Tests if all propagation functions registered with the queue have ceased to exist.

incAPF

```
void incAPF(void);
```

Increments the registration counter by 1.

decAPF

```
void decAPF(void);
```

Decrements the registration counter by 1.

reset

```
void reset(void);
```

Resets the queue. (???)

getSize

```
int getSize(void);
```

Returns the number of queued propagation function entry indicies.

4.7 The class `PEL_FSetProfile`

constructor **PEL_FSetProfile**

```
PEL_FSetProfile(void);
```

Constructs a profile for finite set constraint.

init

```
void init(OZ_FSetConstraint &fset);
```

Initializes the profile with `fset`.

isTouched

```
int isTouched(OZ_FSetConstraint &fset);
```

Tests if the constraint `fset` is more constrained than the constraint, the profile has been initialized with.

isTouchedSingleValue

```
int isTouchedSingleValue(OZ_FSetConstraint &fset);
```

Tests if the constraint `fset` has become a single value since the last initialization of the profile.

isTouchedLowerBound

```
int isTouchedLowerBound(OZ_FSetConstraint &fset);
```

Tests if the lower bound of the constraint `fset` has been further constrained since the last initialization of the profile.

isTouchedUpperBound

```
int isTouchedUpperBound(OZ_FSetConstraint &fset);
```

Tests if the upper bound of the constraint `fset` has been further constrained since the last initialization of the profile.

4.8   The class `PEL_FSetEventLists`

getLowerBound

```
PEL_EventList &getLowerBound(void);
```

Returns the event list for lower bound events.

getUpperBound

```
PEL_EventList &getUpperBound(void);
```

Returns the event list for upper bound events.

getSingleValue

```
PEL_EventList &getSingleValue(void);
```

Returns the event list for single value events.

gc

```
void gc(void);
```

Performs a garbage collection. Has to be called if the hosting propagation is garbage collected.

4.9   The class `PEL_FDProfile`

constructor **PEL_FDProfile**

```
PEL_FDProfile(void);
```

Constructs a profile for finite domain constraint.

init

```
void init(OZ_FDConstraint &fd);
```

Initializes the profile with `fd`.

isTouched

```
int isTouched(OZ_FDConstraint &fd);
```

Tests if the constraint `fd` is more constrained than the constraint, the profile has been initialized with.

isTouchedWidth

```
int isTouchedWidth(OZ_FDConstraint &fd);
```

Tests if the width of the constraint `fd` has been further constrained since the last initialization of the profile.

isTouchedLowerBound

```
int isTouchedLowerBound(OZ_FDConstraint &fd);
```

Tests if the lower bound of the constraint `fd` has been further constrained since the last initialization of the profile.

isTouchedUpperBound

```
int isTouchedUpperBound(OZ_FDConstraint &fd);
```

Tests if the upper bound of the constraint `fd` has been further constrained since the last initialization of the profile.

isTouchedBounds

```
int isTouchedBounds(OZ_FDConstraint &fd);
```

Tests if at least one of the bounds of the constraint `fd` has been further constrained since the last initialization of the profile.

isTouchedSingleValue

```
int isTouchedSingleValue(OZ_FDConstraint &fd);
```

Tests if the constraint `fd` has become a single value since the last initialization of the profile.

4.10 The class `PEL_FDEventLists`

getBounds

```
PEL_EventList &getBounds(void);
```

Returns the event list for bound events.

getSingleValue

```
PEL_EventList &getSingleValue(void);
```

Returns the event list for single value events.

gc

```
void gc(void);
```

Performs a garbage collection. Has to be called if the hosting propagation is garbage collected.

4.11 The class `PEL_SuspVar`

This class defines the minimal functionality required by classes derived from `PEL_SuspVar`.

wakeup

```
virtual int wakeup(void) = 0;
```

This function is required to be defined the derived classes.

4.12 The class `PEL_SuspFSetVar`

constructor `PEL_SuspFSetVar`

```
                    PEL_SuspFSetVar(void);
```

Constructs an uninitialized library finite set variable.

constructor `PEL_SuspFSetVar`

```
              PEL_SuspFSetVar(PEL_FSetProfile &fsetp,
                              OZ_FSetConstraint &fset,
                              PEL_FSetEventLists &fsetel,
                              PEL_PropQueue &pq,
                              PEL_PropFnctTable &pft,
                              int first = 1);
```

Constructs an initialized library finite set variable which is directly connected with the corresponding variable in the constraint store.

constructor `PEL_SuspFSetVar`

```
              PEL_SuspFSetVar(OZ_FSetConstraint &fsetl,
                              PEL_FSetEventLists &fsetel,
                              PEL_PropQueue &pq,
                              PEL_PropFnctTable &pft);
```

Constructs an initialized library finite set variable which is not directly connected with the corresponding variable in the constraint store. This constructor is used if the library variable is subordinated to the store variable, e.g. when implementing a clause of a disjunction.

`init`

```
              PEL_SuspFSetVar * init(PEL_FSetProfile &fsetp,
                                     OZ_FSetConstraint &fset,
                                     PEL_FSetEventLists &fsetel,
                                     PEL_PropQueue &pq,
                                     PEL_PropFnctTable &pft,
                                     int first = 1);
```

This initialization function is associated with the constructor for the directly connected library variable and returns a pointer the library variable.

`init`

```
              PEL_SuspFSetVar * init(OZ_FSetConstraint &fsetl,
                                     PEL_FSetEventLists &fsetel,
                                     PEL_PropQueue &pq,
                                     PEL_PropFnctTable &pft);
```

This initialization function is associated with the constructor for the not directly connected library variable and returns a pointer the library variable.

`propagate_to`

```
          int propagate_to(OZ_FSetConstraint &fset, int first = 0);
```

The constraint `fset` is propagated to the library variable and `wakeup` is called if necessary. The function returns 0 in case propagation fails. Otherwise it returns 1.

wakeup

```
virtual int wakeup(int first = 0);
```

Causes propagation functions to be scheduled for rerun according to the constraints imposed on this variable since the last invocation of this function. This function returns 1 if variable denotes a single value and else 0.

operator *

```
OZ_FSetConstraint &operator * (void);
```

Returns the finite set constraint associated with this variable.

operator ->

```
OZ_FSetConstraint * operator -> (void);
```

Returns the pointer to the finite set constraint associated with this variable.

4.13 The class `PEL_SuspFDIntVar`

constructor **PEL_SuspFDVar**

```
PEL_SuspFDVar(void);
```

Constructs an uninitialized library finite set variable.

constructor **PEL_SuspFDVar**

```
PEL_SuspFDIntVar(PEL_FDProfile &fdp,
                 OZ_FiniteDomain &fdv,
                 PEL_FDEventLists &fdel,
                 PEL_PropQueue &pd,
                 PEL_PropFnctTable &pft,
                 int first = 1);
```

Constructs an initialized library finite domain variable which is directly connected with the corresponding variable in the constraint store.

constructor **PEL_SuspFDVar**

```
PEL_SuspFDIntVar(OZ_FiniteDomain &fdl,
                 PEL_FDEventLists &fdel,
                 PEL_PropQueue &pd,
                 PEL_PropFnctTable &pft);
```

Constructs an initialized library finite domain variable which is not directly connected with the corresponding variable in the constraint store. This constructor is used if the library variable is subordinated to the store variable, e.g. when implementing a clause of a disjunction.

init

```
PEL_SuspFDIntVar * init(PEL_FDProfile &fdp,
                        OZ_FiniteDomain &fd,
                        PEL_FDEventLists &fdel,
                        PEL_PropQueue &pq,
                        PEL_PropFnctTable &pft,
                        int first = 1);
```

This initialization function is associated with the constructor for the directly connected library variable and returns a pointer the library variable.

init

```
PEL_SuspFDIntVar * init(OZ_FiniteDomain &fdl,
                        PEL_FDEventLists &fdel,
                        PEL_PropQueue &pq,
                        PEL_PropFnctTable &pft);
```

This initialization function is associated with the constructor for the not directly connected library variable and returns a pointer the library variable.

propagate_to

```
int propagate_to(OZ_FiniteDomain &fd, int first = 0);
```

The constraint `fd` is propagated to the library variable and `wakeup` is called if necessary. The function returns 0 in case propagation fails. Otherwise it returns 1.

wakeup

```
virtual int wakeup(int first = 0);
```

Causes propagation functions to be scheduled for rerun according to the constraints imposed on this variable since the last invocation of this function. This function returns 1 if variable denotes a single value and else 0.

operator *

```
OZ_FiniteDomain &operator * (void);
```

Returns the finite domain constraint associated with this variable.

operator ->

```
OZ_FiniteDomain * operator -> (void);
```

Returns the pointer to the finite domain constraint associated with this variable.

Index

LP
    config