The Mozart Constraint Extensions Tutorial

Tobias Müller

Abstract

This tutorial provides the knowledge to go beyond the built-in constraint capabilities of Mozart Oz 3. This tutorial is complemented by *"The Mozart Constraint Extensions Reference"*.

Motivation A major design goal of Oz is to provide for a wide range of applications the right level of programming abstractions. Though Mozart Oz 3 features a full-fledged finite domain and finite set constraint solver providing for the functionality required to solve combinatorial problems efficiently, it is often desirable to implement constraints in C++. There may be several reasons to do so, as for example, that a given algorithm requires destructively updateable data structures or an already existing C++ library shall be used. Consequently, we opened the constraint solver of Mozart Oz 3 by adding a C/C++ interface for implementing so-called *constraint propagators*. Hereby, a constraint propagator is the implementation of a constraint. Further, it may be desirable to have a constraint system available that is not provided by Mozart Oz 3 and one wants to implement it from scratch. Even such cases can be handled by the C/C++ interface. Finally, the integration of linear (integer) programming solvers is explained which are standard means in Operations Research to tackle certain combinatorial problem classes.

Structure of the Manual The user manual consists of three parts:

1. The first part explains how to implement various propagators. It starts with a propagator for the constraint $x + y = z$ over finite domains and introduces the tools and techniques needed. This propagator will be refined such that it is able to detect equal variables and reduces to a more specialised propagator. Then a functionally nestable version of the addition propagator will be derived. We go on with a propagator that can deal with vectors of variables. As example serves the so-called *element* constraint. The implementation of a propagator using finite set and finite domain constraints is explained next. Finally more advanced topics, like the implementation of reified constraints, are discussed.
   Note that it is not the intention of this manual to provide sophisticated algorithms.

2. The second part explains the implementation of constraint systems from scratch, i.e., not only the propagators of a certain constraint system but also the basic constraints. As example, constraints over real intervals are implemented.

3. The third part explains the integration and usage of linear programming solvers like CPLEX [6] and LP_SOLVE [4] from within Mozart Oz 3. To demonstrate the benefits of jointly using propagation-based and linear programming-based solvers knapsack problems are tackled.

Prerequisites The reader is supposed to have a working knowledge in the C/C++ programming language and to be familiar with constraint-based problem solving techniques in Oz. An excellent supplementary text book on C++ is [9]. Constraint-based problem solving techniques in Oz are explained in *"Finite Domain Constraint Programming in Oz. A Tutorial."* resp. *"Problem Solving with Finite Set Constraints in Oz. A Tutorial."*. The CPI uses the *native functor* interface of Mozart Oz 3. Have a look at Part *Native C/C++ Extensions, (Application Programming)* resp. *"Interfacing to C and C++"* for details.

Credits

Mozart logo by Christian Lindig

License Agreement

This software and its documentation are copyrighted by the German Research Center for Artificial Intelligence (DFKI), the Swedish Institute of Computer Science (SICS), and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Contents

Implementing Propagators

1.1 Basic Concepts

This chapter explains the basic concepts of computation with finite domain constraints in Oz. Further, it explains the implementation of constraints by propagators. For more details see *"Finite Domain Constraint Programming in Oz. A Tutorial."*.

1.1.1 Computation with Constraints in Oz

Basic Constraint   A *Basic Constraint* takes the form $x = n, x = y$ or $x \in D$, where x and y are variables, n is a non-negative integer and D is a finite domain.

Constraint Store   The basic constraints reside in the *Constraint Store*. Oz provides efficient algorithms to decide satisfiability and entailment for basic constraints.

Propagators   For more expressive constraints, like $x + y = z$, deciding their satisfiability is not computationally tractable. Such non-basic constraints are not contained in the constraint store but are imposed by *propagators*. A propagator is a *computational agent* which is *imposed on* the variables occurring in the corresponding constraint. These variables are called the propagator's *parameters*. The propagator tries to narrow the domains of the variables it is imposed on by amplifying the store with basic constraints.

Constraint Propagation   This narrowing is called *constraint propagation*. A propagator P amplifies the store S by writing a basic constraint ϕ to it, if $P \wedge S$ entails ϕ but S on its own does not. If P ceases to exist, it is either entailed by the store S, or $P \wedge S$ is unsatisfiable. Note that the amount of propagation depends on the operational semantics of the propagator.

As an example, assume a store containing $x, y, z \in \{1, \ldots, 10\}$. The propagator for $x + y < z$ narrows the domains to $x, y \in \{1, \ldots, 8\}$ and $z \in \{3, \ldots, 10\}$ (since the other values cannot satisfy the constraint). Adding the constraint $z = 5$ causes the propagator to strengthen the store to $x, y \in \{1, \ldots, 3\}$ and $z = 5$. Imposing $x = 3$ lets the propagator narrow the domain of y to 1. We say that the propagator $x + y < z$ *constrains* the variables x, y and z.

Computation Space   Computation in Oz takes place in so-called *computation spaces*. For the purpose of the tutorial it is sufficient to consider a computation space as consisting of the *constraint store* and *propagators* connected to the store.

1.1.2 Implementation of Propagators

The computational model sketched in Section 1.1.1 is realised by the Oz runtime system, which is implemented by an abstract machine [7], called the *emulator*. In this section, the internal structure of propagators and their interaction with the emulator is explained.

A propagator exists in different execution states and has to be provided with resources like computation time and heap memory by the emulator. A propagator synchronises on the constraint store and may amplify it with basic constraints.

A propagator *reads* the basic constraints of its parameters. In the process of constraint propagation it *writes* basic constraints to the store.

The emulator *resumes* a propagator when the store has been amplified in a way the propagator is waiting for. For example, many propagators will be resumed only in case the bounds of a domain have been narrowed.

Handling Propagators   A propagator is created by the execution of an Oz program. To resume a propagator if one of its parameters is further constrained, one has to attach the propagator somehow to the parameters. To this end, a reference to the propagator is added to so-called *suspension lists* of the propagator's parameters; we say, a propagator is *suspending* on its parameters.

A resumed propagator returns a value of the predefined type `OZ_Return`:

```
enum OZ_Return {OZ_ENTAILED, OZ_FAILED, OZ_SLEEP}
```

In order to schedule propagators, the emulator maintains for each propagator an execution state, namely `running`, `runnable`, `sleeping`, `entailed`, and `failed`. The emulator's scheduler switches a propagator between the execution states as shown in Figure 1.1.

When a propagator is created, its state is immediately set `running` and the scheduler allocates a time slice for its first run. After every run, when the constraint propagation was performed, the emulator evaluates the propagator's return value. The value `OZ_FAILED` is returned if the propagator (according to its operational semantics) detects its inconsistency with the store. The emulator sets the propagator's execution state to `failed` and the computation is aborted. The propagator will be ignored by the emulator until it is eventually disposed by the next garbage collection.

The return value `OZ_ENTAILED` indicates that the propagator has detected its entailment by the constrained store, i.e. it cannot further amplify the constraint store. The emulator sets the propagator's execution state to `entailed`. It happens the same as for a failed propagator: it will be ignored until it is disposed by garbage collection.

If the propagator can neither detect inconsistency nor entailment, it returns `OZ_SLEEP`. Because the propagator may amplify the store in the future, it remains in the suspension lists. Its execution state is set to `sleeping`.

Figure 1.1 Propagator states and the transitions between them
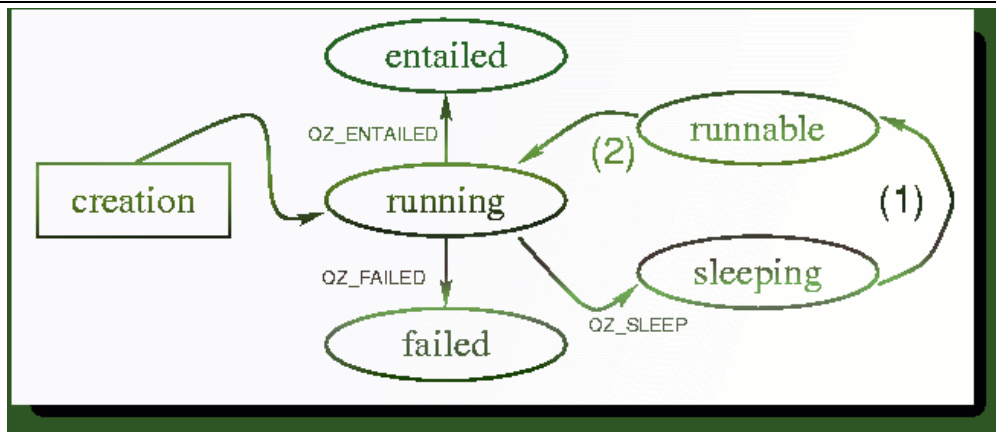
A propagator is resumed when the domain of at least one of its parameters is further narrowed. In this case, the emulator scans the suspension list of that variable and either deletes entries where the propagator's execution state is `failed` resp. `entailed` or switches the execution state of the corresponding propagator to `runnable`. This is indicated by transition (1) in Figure 1.1. Now, the scheduler takes care of the propagator and will schedule it later on (the transition (2) from `runnable` to `running` is subject to the scheduler's policy and will be not discussed here).

The parameters of a propagator are stored in its state. Hence, reading and writing of basic constraints is done by the propagator itself. If a propagator constrains a variable according to its operational semantics, it informs the emulator that the corresponding suspension lists have to be scanned.

1.2 Getting Started

This section makes the reader familiar with the CPI by implementing a propagator for the constraint $x + y = z$ and explains the steps to be taken to get it running.

1.2.1 Prerequisites

The implementation of new propagators via the CPI requires a correctly installed Oz system. The following points should be obeyed.

Include File. To obtain the functionality provided by the CPI include the file `mozart_cpi.hh` in the appropriate C/C++ source files.

Platform-independent compilation and linkage of native functors. Use `oztool` to compile C/C++ source programs (`oztool c++`) and to link object files (`oztool ld`). It has the right options resp. search paths set depending on your current platform and environment. See Section *Compilation*, *(Application Programming)* for details on `oztool`.

Naming Conventions. Identifiers starting with `oz_` are provided by the CPI and must not be defined by the programmer.

1.2.2   Building a Propagator

This section explains by means of an example the constraint propagator interface of
Oz. We implement the propagator for the constraint $x + y = z$. For the sake of clarity
we use a rather straightforward algorithm here. The operational semantics will provide
as much pruning/propagation as possible. This is in contrast to the constraint $x + y = z$
supplied by the finite domain library (see Section *Miscellaneous Propagators*, *(System
Modules)*), which reasons only over bounds of domains.

1.2.2.1   A Propagator's Class Definition

CPI class `OZ_Propagator`   The emulator requires a uniform way to refer to all in-
stances of propagators. This is realised by providing the class `OZ_Propagator`, which
is the class all propagators have to be inherited from. Therefore, the emulator can refer
to any propagator by a pointer of type `(OZ_Propagator *)` The class `OZ_Propagator`
is in terms of C++ a so-called *abstract base class*, i.e. no object of such a class can
be created (since for at least one member function intentionally no implementation is
provided, indicated by an appended `=0`. Instead, it defines the minimal functionality
required of all classes inherited from it. The following code depicts a fragment of the
definition of the class `OZ_Propagator` defined by the CPI (in the file `mozart_cpi.hh`).
It shows all member functions which have to be defined in a derived class.

```cpp
class OZ_Propagator {
public:
  OZ_Propagator(void);
  virtual OZ_Term getParameters(void) const      = 0;
  virtual size_t sizeOf(void)                     = 0;
  virtual void gCollect(void)                     = 0;
  virtual void sClone(void)                       = 0;
  virtual OZ_Return propagate(void)               = 0;
  virtual OZ_PropagatorProfile * getProfile(void) const = 0;
}
```

There are basically three groups of member functions dealing with reflection, memory
management, and constraint propagation. Member functions concerned with reflection
allow to obtain information about a certain instance of a propagator. For example, this
is used to generate a message in case of a top-level failure.

`getProfile()`   For each propagator class, one instance of class `OZ_PropagatorProfile`
is allocated. This class is intended to give the Oz Profiler access to some information
about this class, for instance a count of the number of invocations of propagators be-
longing to this class. This function must return a pointer to this instance, but otherwise
the programmer needs not to be concerned about it. Note that for the profile function
to be shared for all instances, it has to be declared `static`.

`getParameters()`   The arguments of a propagator are returned by `getParameters()`
as a list represented as an Oz heap data structure. This is denoted by the return type
`OZ_Term`.

sizeOf() Memory management of Oz requires to know the size of a propagator. The member function sizeOf() implements this functionality. Its return type is defined in the standard header **<**stddef**.h>**.

gCollect() and **sClone()** Further, on garbage collection and space cloning references into heap which are held in the state of the propagator (or somehow reachable by a propagator) have to be updated, since items stored on the heap are occasionally moved to a new location. The member functions gcollect() and sClone are provided for that purpose. The definition of these functions is identical for most propagators. For an example where the difference of both functions matters see Section 1.7.2.

propagate() The most important member function is propagate(). It is responsible for the actual constraint propagation and is called by the emulator when the propagator's execution state is switched to running. The returned value of type OZ_Return indicates the runtime system the outcome of the propagation performed by propagate().

The implementation of the addition propagator starts with the definition of the class AddProp. The definition of the member function propagate() is explained in Section 1.2.2.2.

```
#ifndef NDEBUG
#include <stdio.h>
#endif

#include "mozart_cpi.hh"

class AddProp : public OZ_Propagator {
  friend OZ_C_proc_interface *oz_init_module(void);
private:
  static OZ_PropagatorProfile profile;
  OZ_Term _x, _y, _z;
public:
  AddProp(OZ_Term a, OZ_Term b, OZ_Term c)
   : _x(a), _y(b), _z(c) {}
  virtual OZ_Return propagate(void);

  virtual size_t sizeOf(void) {
    return sizeof(AddProp);
  }
  virtual void gCollect(void) {
    OZ_gCollectTerm(_x);
    OZ_gCollectTerm(_y);
    OZ_gCollectTerm(_z);
  }
  virtual void sClone(void) {
    OZ_sCloneTerm(_x);
    OZ_sCloneTerm(_y);
    OZ_sCloneTerm(_z);
```

```
    }
    virtual OZ_Term getParameters(void) const {
      return OZ_cons(_x,
                     OZ_cons(_y,
                             OZ_cons(_z,
                                     OZ_nil())));
    }
    virtual OZ_PropagatorProfile *getProfile(void) const {
      return &profile;
    }
};


OZ_PropagatorProfile AddProp::profile;
```

The propagator stores in its state, i.e. in its data members, references to the variables it is imposed on (namely `_x`, `_y` and `_z` of type `OZ_Term`. The constructor of the class `AddProp`, which is invoked by the header function, initialises the data members with the arguments of the corresponding Oz application. The member function `sizeOf()` returns the number of bytes occupied by the addition propagator using C/C++'s `sizeof` operator. The CPI provides for the functions `OZ_gCollectTerm()` and `OZ_sCloneTerm()`, which are used for the implemention of the member functions `gcollect()` and `sClone()`, which apply `gCollectTerm()` resp. `sCloneTerm()` to all data members of type `OZ_Term`. The construction of lists is supported by the interface abstractions `OZ_cons()` and `OZ_nil()` (see Section *Term access and construction*, *(Interfacing to C and C++)*). The function `getParameters()` straightforwardly composes a list containing the references to the arguments hold in the propagator's state. The reason for the friend declaration will become clear in Section 1.2.2.3.

1.2.2.2 The Propagation Part of a Propagator

The member function `propagate()` implements the algorithm which defines the operational semantics of the propagator, i.e. the amount of constraint propagation achieved at each invocation.

The algorithm used here rebuilds the domains of the variables always from scratch. Therefore, auxiliary domains for each variable are introduced which are initially empty. For all values of the domains of *x* and *y* it is checked if there is a consistent value in the domain of *z*. If so, the values are added to the corresponding auxiliary domains. Finally, the domains of the variables are constrained,i.e. intersected, with the corresponding auxiliary domains. Consequently, the core of the program code consists of two nested loops iterating over all values of the domains of *x* and *y*.

```
#define FailOnEmpty(X) if((X) == 0) goto failure;


OZ_Return AddProp::propagate(void)
{

  OZ_FDIntVar x(_x), y(_y), z(_z);
```

```
        OZ_FiniteDomain x_aux(fd_empty),
                        y_aux(fd_empty),
                        z_aux(fd_empty);

    for (int i = x->getMinElem(); i != -1;
         i = x->getNextLargerElem(i))
      for (int j = y->getMinElem(); j != -1;
           j = y->getNextLargerElem(j))
        if (z->isIn(i + j)) {
          x_aux += i;
          y_aux += j;
          z_aux += (i + j);
        }

    FailOnEmpty(*x &= x_aux);
    FailOnEmpty(*y &= y_aux);
    FailOnEmpty(*z &= z_aux);

    return (x.leave() | y.leave() | z.leave())
      ? OZ_SLEEP : OZ_ENTAILED;

failure:
  x.fail();
  y.fail();
  z.fail();
  return OZ_FAILED;
}
```

CPI class `OZ_FDIntVar`   A propagator needs direct access to the variables it is imposed on. The interface class `OZ_FDIntVar` provides member functions to access variables in the constraint store. The constructor dereferences a variable in the store and stores the dereferenced information in the state of the newly created object. The operators `*` and `->` are overloaded to provide direct access to the finite domain of a variable in the store or to invoke member functions of the class `OZ_FiniteDomain` (see below).

CPI class `OZ_FiniteDomain`   The finite domain of a variable is represented by an instance of the class `OZ_FiniteDomain`, modifying their value is immediately visible in the constraint store. Calling the constructor with the value `fd_empty` creates an empty finite domain, as used for the auxiliary variables here. The operator `+=` adds a value to a domain. The operator `&=` intersects two domains, modifies the domain on the left hand side and returns the size of the intersected domain. The member function `getMinElem()` returns the smallest value of the domain and `getNextLargerElem(i)` returns the smallest value of the domain larger than `i` (both return `-1` when they reach their respective end of the domain). Testing whether a value is contained in a domain or not can be done by the member function `isIn()`.

The implementation   The implementation of the constraint $x + y = z$ proceeds as follows. First the variables in the store are retrieved and stored in the local C/C++ variables `x`, `y` and `z`. The corresponding auxiliary domains are held in the variables `x_aux`, `y_aux` and `z_aux`, which are initialised to empty domains. Two nested for-loops enumerate all possible pairs (v_x, v_y) of values of the domains of x and y. Each loop starts from the smallest value of the domain and proceeds until -1 is returned, indicating that there is no larger value. If there is a value v_z in the domain of z satisfying the relation $v_x + v_y = v_z$, these values are added to the appropriate auxiliary domains. After completing the nested loops, the domains of the variables are constrained by intersecting them with the auxiliary domains.

FailOnEmpty()   The macro `FailOnEmpty()` branches to the label `failure` if its argument results in the value 0. Thereby, constraining the domain of a variable to an empty domain causes the execution to branch to label `failure` and eventually to return `OZ_FAILED` to the emulator. The return value of the member function `leave()` of class `OZ_FDIntVar` is used to decide whether the propagator returns `OZ_SLEEP` or `OZ_ENTAILED`. The return value `OZ_ENTAILED` indicates entailment and is returned if all variable's domains are singletons. Otherwise, `OZ_SLEEP` is returned and the propagator is resumed when at least one of its variables is constrained again.

Before leaving `propagate()`, the member function `leave()` has to be called. If the variable's domain has been constrained by the propagator, it causes the scheduler to switch all propagators waiting for further constraints on that variable to become `runnable`. The return value of `leave` is 0 if the domain became a singleton, otherwise 1. This information is used to decide whether a propagator is entailed or not. In case the propagator encounters an empty domain or any other inconsistency, the member function **fail**`()` has to be called to do some cleanups before `propagate()` is left.

1.2.2.3   Creating a Propagator

The header function   Before a propagator can be created and introduced to the emulator, its variables must be sufficiently constrained, e.g. the variables must be constrained to finite domains. In case, only a subset of variables is sufficiently constrained, the computation will suspend and resume again when more constraints become available. This is checked by the header function, which is called by the runtime system, when an appropriately connected Oz abstraction is applied. For our example, this function is called `fd_add`.

Determining when to resume a propagator   Further, when a propagator is imposed on a variable, it has to be determined which changes to the domain resume the propagator again. The alternatives are to resume a propagator if the variable's domain becomes a singleton, the bounds are narrowed or some value is removed from the domain.

The macros `OZ_C_proc_begin` and `OZ_C_proc_end` are provided to allow the implementation of C/C++ functions which are compliant with the calling conventions of Oz's emulator.

The first argument of the macro `OZ_C_proc_begin` defines the name of the function and the second argument the number of arguments of type `OZ_Term`. The macro

`OZ_args` provides access to the actual argument. The name of the function has to obey certain rules to be compatible with the module `Foreign` which enables linking object files to a running Oz runtime system. The definition of the macro `OZ_EXPECTED_TYPE` is explained in Section *Macros*, *(The Mozart Constraint Extensions Reference)*.

```
OZ_BI_define(fd_add, 3, 0)
{
  OZ_EXPECTED_TYPE(OZ_EM_FD","OZ_EM_FD","OZ_EM_FD);

  OZ_Expect pe;

  OZ_EXPECT(pe, 0, expectIntVar);
  OZ_EXPECT(pe, 1, expectIntVar);
  OZ_EXPECT(pe, 2, expectIntVar);

  return pe.impose(new AddProp(OZ_in(0),
                               OZ_in(1),
                               OZ_in(2)));
}
OZ_BI_end
```

Using `OZ_EXPECT`   The macro `OZ_EXPECT` ensures that incompatible constraints on the propagator's parameters lead to failure and insufficient constraints cause the execution to be suspended until more constraints become known. An object of class `OZ_Expect` collects in its state all variables the propagator is to be imposed on. Such an object is at the first argument position of `OZ_EXPECT`. The second argument of `OZ_EXPECT` determines which argument of `fd_add` shall be checked. The member function `expectIntVar()` of class `OZ_Expect` expects a variable to be already constrained to a finite domain. If a variable is sufficiently constrained, it is stored in the state of the object `pe`. The second argument of `expectIntVar` is used to determine what kind of domain pruning causes a propagator to be resumed. Its default value is `fd_prop_any`, i.e. a propagator is resumed on any pruning of the domain. For further details see Section 1.4.

Creation of a propagator   Finally, the actual propagator is created by calling its constructor via the application of the `new` operator. The reference to the newly created propagator is passed as argument to `impose()`, a member function of `OZ_Expect`, which executes the `propagate()` method and introduces the propagator to the emulator.

Connecting Propagators and Oz Code   Propagators are connected with Mozart Oz 3 as native functors according to Section *Deployment*, *(Application Programming)*. To enable that one has to define a function `oz_init_module`.

```
OZ_BI_proto(fd_add);

OZ_C_proc_interface *oz_init_module(void)
```

```
{
  static OZ_C_proc_interface i_table[] = {
    {"add", 3, 0, fd_add},
    {0,0,0,0}
  };

  AddProp::profile      = "addition/3";

  printf("addition propagator loaded\n");
  return i_table;
}
```

The line `AddProp::profile = "addition/3";` assigns explicitly the name `"addition/3"` to the propagator. The default name is `"<anonymous propagator>"`.

Before a native functor can be loaded it must be compiled according to Section *Compilation*, *(Application Programming)*. Supposing the C/C++ code is stored in the file `ex_a.cc`, then the following lines create the object file.

```
oztool c++ -c ex_a.cc -o ex_a.o
oztool ld  -o ex_a.so-linux-i486 ex_a.o
```

The Oz code below loads the object file `ex_a.so-linux-i486` and makes the Oz abstraction `FD_PROP.add` available. The procedure `FD_PROP.add` takes 3 arguments and imposes the addition propagator implemented in the sections before.

```
declare FD_PROP
local
   FD_PROP_O = {{New Module.manager init}
               link(url: 'ex_a.so{native}' $)}
in
   FD_PROP = fd(add: FD_PROP_O.add)
   {Browse FD_PROP}
end
```

After feeding in the above Oz code the addition propagator is available and can be used. To do so feed the following code in line by line. The results are shown in the Oz browser (shown in comments appended to lines).

```
declare X Y Z in
{Browse [X Y Z]}    % [X Y Z]

[X Y Z] ::: 0#10    % [X{0#10} Y{0#10} Z{0#10}]

{FD_PROP.add X Y Z} % [X{0#10} Y{0#10} Z{0#10}]

X :: [1 3 5 7 9]    % [X{1 3 5 7 9} Y{0#9} Z{1#10}]
Y :: [1 3 5 7 9]    % [X{1 3 5 7 9} Y{1 3 5 7 9}
```

```
                                  %   Z{2 4 6 8 10}]
          Y <: 5                  % [X{1 3 5 7 9} Y{1 3}
                                  %   Z{2 4 6 8 10}]
          Y \=: 3                 % [X{1 3 5 7 9} 1 Z{2 4 6 8 10}]
```

Troubleshooting   Debugging a propagator is usually done by `gdb` [10] in conjunction with `emacs` [11]. The Oz Programming Interface provides adequate means to support debugging based on these two tools. We refer the reader to Section *Running under gdb*, *(The Oz Programming Interface)* for details.

1.3   Replacing a Propagator

There are situations when a propagator should be replaced by another one. The replacing propagator must have the same declarative semantics, but should provide a more efficient implementation for a particular situation.

Consider the following situation: First a propagator $x + y = z$ was imposed. At a later point in time the constraint $x = y$ is told to the constraint store. The equality constraint allows to replace $x + y = z$ by $2x = z$. The rules below show how $x + y = z$ can be replaced by another (equality) constraint, if two variables are set equal.

Rule 1: $x + y = z \land x = y \rightarrow z = 2x$

Rule 2: $x + y = z \land x = z \rightarrow y = 0$

Rule 3: $x + y = z \land y = z \rightarrow x = 0$

Such simplifications can be implemented by replacing a propagator by another one. The CPI provides for that purpose in `OZ_Propagator` a group of member functions `replaceBy`. This section demonstrates how to realise the above simplifications using the example of the previous section.

1.3.1   A Twice Propagator

The implementation of the simplification rule $x + y = z \land x = y \rightarrow 2x = z$ requires a propagator for the constraint $2x = z$. The following code defines the class `TwiceProp`.

```
class TwiceProp : public OZ_Propagator {
private:
  static OZ_PropagatorProfile profile;
  OZ_Term _x, _z;
public:
  TwiceProp(OZ_Term a, OZ_Term b)
   : _x(a), _z(b) {}
  virtual OZ_Return propagate(void);

  virtual size_t sizeOf(void) {
```

```
      return sizeof(TwiceProp);
    }
    virtual void gCollect(void) {
      OZ_gCollectTerm(_x);
      OZ_gCollectTerm(_z);
    }
    virtual void sClone(void) {
      OZ_sCloneTerm(_x);
      OZ_sCloneTerm(_z);
    }
    virtual OZ_Term getParameters(void) const {
      return OZ_cons(_x,
                     OZ_cons(_z,
                             OZ_nil()));
    }
    virtual OZ_PropagatorProfile *getProfile(void) const {
      return &profile;
    }
};


OZ_PropagatorProfile TwiceProp::profile;
```

The member function `propagate()` mainly consists of a **for**-loop collecting in auxiliary variables the values v_x and v_z satisfying the relation $2v_x = v_z$.

```
OZ_Return TwiceProp::propagate(void)
{
  OZ_FDIntVar x(_x), z(_z);

  OZ_FiniteDomain x_aux(fd_empty), z_aux(fd_empty);

  for (int i = x->getMinElem(); i != -1;
       i = x->getNextLargerElem(i)) {
    int i2 = 2 * i;
    if (z->isIn(i2)) {
      x_aux += i; z_aux += i2;
    }
  }

  FailOnEmpty(*x &= x_aux);
  FailOnEmpty(*z &= z_aux);

  return (x.leave() | z.leave())
    ? OZ_SLEEP : OZ_ENTAILED;

failure:
  x.fail(); z.fail();
  return OZ_FAILED;
}
```

1.3.2 Equality Detection and Replacement

Imposing equality on variables is done by unification. A propagator is *always* resumed if at least one variable of its parameters is unified with another variable. The class `OZ_Propagator` provides for a member function `mayBeEqualVars()`, which returns 1 in case the propagator is resumed because at least one of its parameters was involved in a unification. Otherwise it returns 0.

To detect if the addition propagator is resumed because of a unification the following macro is defined. First, it checks if the propagator's parameters were involved in some unification. If that is the case, all possible combinations of equated variables are tested. The CPI function `OZ_isEqualVars()` is provided for that purpose. It takes two heap references and returns 1 if they refer to the same variable. In case equal variables are detected the execution branches to a `return` statement, which returns the value produced by executing the function passed as argument of the macro.

```
#define ReplaceOnUnify(EQ01, EQ02, EQ12) \
  if (mayBeEqualVars()) {                 \
    if (OZ_isEqualVars(_x, _y)) {         \
      return (EQ01);                      \
    }                                     \
    if (OZ_isEqualVars(_x, _z)) {         \
      return (EQ02);                      \
    }                                     \
    if (OZ_isEqualVars(_y, _z)) {         \
      return (EQ12);                      \
    }                                     \
  }
```

The macro is inserted as first statement in the code of the addition propagator. The member functions `replaceBy()` and `replaceByInt()` provided by `OZ_Propagator` replace the addition propagator according to their arguments by another propagator or a basic constraint.

```
OZ_Return AddProp::propagate(void)
{

  ReplaceOnUnify(replaceBy(new TwiceProp(_x, _z)),
                 replaceByInt(_y, 0),
                 replaceByInt(_x, 0));
```

The first argument of the macro causes the addition propagator to be replaced by the twice propagator, which implements reduction rule 1 (page 11). The member function `replaceBy()` expects a pointer to a propagator which is generated by applying the `new` operator to the constructor of the class `TwiceProp`. The second and third macro argument realize the simplification rules 2 (page 11) and 3 (page 11) by imposing the constraint $y = 0$ resp. $x = 0$.

1.3.3   Benefits of Replacing Propagators

In most of the cases when propagators are replaced the execution becomes faster without obtaining a stronger propagation simply by the fact that redundant computation is avoided. The example of this section provides for even better propagation by imposing a stronger constraint. This can be observed when running the following Oz code. Of course, the updated module has to be loaded before.

```
declare X Y Z in
{Browse [X Y Z]}      % [X Y Z]

X :: [1 3 5 7 9]      % [X{1 3 5 7 9}
Y :: [1 3 5 7 9]      %  Y{1 3 5 7 9}
Z :: 0#10             %  Z{0#10}]

{FD_PROP.add X Y Z} % [X{1 3 5 7 9} Y{1 3 5 7 9}
                      %  Z{2 4 6 8 10}]
X = Y                 % [Y{1 3 5} Y{1 3 5} Z{2 6 10}]
```

Note that the constraint $x = y$ causes $x + y = z$ to be replaced by $2x = z$, so that the domain of x and y is further constrained to $\{1, 3, 5\}$, which is not the case for the propagator implemented in Section 1.2.

1.4   Imposing Propagators

The CPI provides a generic way to implement different schemes for imposing propagators. This section discusses the following issues:

- How to implement a nestable propagator.

  It is explained how to make the addition propagator of Section 1.2 nestable.

- How to extend the class `OZ_Expect` to cope with structured parameters.

The answers to these questions will be used in later sections, for example, when we come to implement propagators imposed on more than just single variables.

1.4.1   Basic Concepts

A propagator is imposed by a C/C++ function, a so-called *header function*, that is connected to an Oz abstraction. The application of such an abstraction results in calling the corresponding header function and finally in imposing the propagator.

CPI class `OZ_Expect` The class `OZ_Expect` provides the functionality to fulfill the tasks mentioned above. It provides member functions to control the imposition of a propagator and to determine the constraints which have to be present in the store before a propagator is imposed.

The class `OZ_Expect` provides a group of member functions to examine the constraints of a propagator's parameters. The names of these member functions begin with `expect`. The basic idea is to define for each parameter a constraint ϕ (expected to be present in the store) in terms of `expect` member functions and to decide whether ϕ is entailed resp. disentailed by the store (by evaluating the return value of the `expect` function expressing the constraint ϕ). If entailment of ϕ for a parameter cannot be decided yet then the constraint in the store for this parameter is *insufficient*.

The type of the return value allows to handle structured parameters. The definition of the return type is as follows.

```
struct OZ_expect_t { int size, accepted; }
```

The meaning of the fields `size` and `accepted` is explained by the following examples.

Example 1. Assume a parameter is expected to be an integer, then the field `size` of the returned value is 1. In case this parameter is currently a variable then the field `accepted` is 0. An inconsistent constraint, like for instance a literal, would be indicated by -1. The value 1 in the field `accepted` for our example means that the examined parameter is an integer.

Example 2. Let us suppose we expect a parameter to be a vector with *n* integer fields, where a vector is either a closed record, a tuple, or a list. First the parameter is expected to be a vector (which is one constraint expected to be found in the store) and then all its *n* elements are to be integers, which determines the field `size` of the return value to be $n+1$. If the field `accepted` is also $n+1$, then all expected constraints are present. Otherwise appropriate action has to be taken, for example, suspending the execution of the header function. The implementation to check for a vector of finite domain variable is discussed in Section 1.4.1.

An instance of the class `OZ_Expect` maintains two sets, namely *A* and *B*. In the course of checking parameters `expect` member functions collect variables in either of these two sets. Variables which are constrained according to the corresponding `expect` function are added to set *A*. All the other variables are added to set *B*. The `expect` function for finite domain variables has an extra argument to determine the event which resumes the propagator, for example only narrowing of the bounds of the domain. This information is maintained in the sets too.

Leaving a header function by calling the member function `suspend` (see Section *Member Functions for Control Purposes*, *(The Mozart Constraint Extensions Reference)*) causes the header function to be resumed if variables collected in set *B* are further constrained.

Calling `OZ_Expect::impose()` introduces the propagator which is passed as argument to the runtime system and makes the propagator resume if at least one variable of both sets is constrained in a way defined by the corresponding `expect` function. Additionally, variables in set *B* are constrained to be finite domain variables. This will be used for the implementation of nestable propagators.

1.4.2 Imposing Nestable Propagators

In Section 1.2.2.3 not too much attention was paid to propagator imposition. Now more details will be given by the example of a nestable propagator. Let us consider the following Oz code.

```
{FD.times {FD.plus U V} {FD.plus X Y} Z}
```

The propagator `FD.plus` is required to be *nestable*, since one of its parameters is syntactically not accessible and cannot be constrained to a finite domain variable by explicit Oz code. The expansion of the above code makes it clear.

```
local A1 A2 in
  {FD.plus U V A1}
  {FD.plus X Y A2}
  {FD.times A1 A2 Z}
end
```

Due to lexical scoping the implicit variables `A1` and `A2` are inaccessible to outside code. Therefore the two `FD.plus` propagators must constrain `A1` and `A2` to finite domain variables before they are imposed. To simplify the implementation of header functions for propagators, the CPI provides three macros.

`OZ_EXPECTED_TYPE(S)`

defines a C/C++ string `S` typically consisting of a number of substrings separated by commas which describe the constraint expected at the corresponding argument position. The first substring corresponds to the first argument with index 0, the second one to the second argument with index 1 and so on. The substrings are used to generate meaningful messages in case an inconsistent constraint is detected. There are predefined macros (starting with `OZ_EM_` defining strings for the possible constraints to be expected. For details see Section *Macros*, *(The Mozart Constraint Extensions Reference)*. The macro `OZ_EXPECTED_TYPE` is required by `OZ_EXPECT` and `OZ_EXPECT_SUSPEND`.

`OZ_EXPECT(O, P, F)`

checks if the argument at position `P` (`P` is a C integer with a value starting from 0) is constrained according to the semantics of `F` (which is an `expect` member function of class `O`). The type of `F` has to be `OZ_Expect_t O::F(OZ_Term)`. The value of `O` has to be an instance of the class `OZ_Expect` or a class inheriting from it. In case an inconsistent or insufficient constraint is detected, the appropriate action is taken [1] and the C/C++ function is left by a `return` statement. Otherwise, the execution proceeds to the next statement in the header function.

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

is similar to `OZ_EXPECT` except for the case that the constraint defined by `F` is currently not yet entailed. Then it increments the value of `SC` which is expect to be of type `int` and proceeds to the next statement in the header function.

---

[1] For example, in case of a detected inconsistency an error message is emitted and the header function returns `OZ_FAILED` to the runtime system.

In Section 1.2.2.3 the macro `OZ_EXPECT` was already used for the non-nestable addition propagator. The macro `OZ_EXPECT_SUSPEND` is provided to implement nestable propagators. Insufficient constraints for a parameter cause it to increment its argument `SC`. Allowing exactly one argument to be insufficiently constrained implements a nestable propagator.

Therefore the header function has to suspend in case more than one parameter is insufficiently constrained. The class `OZ_Expect` therefore provides the member function `suspend()` which expects a value of type `OZ_Thread`. Details on how to create a thread for a C/C++ function can be found in Section *Threads, (Interfacing to C and C++)*.

```
OZ_BI_define(fd_add_nestable, 3, 0)
{
  OZ_EXPECTED_TYPE(OZ_EM_FD","OZ_EM_FD","OZ_EM_FD);

  OZ_Expect pe;
  int susp_count = 0;

  OZ_EXPECT_SUSPEND(pe,0,expectIntVar,susp_count);
  OZ_EXPECT_SUSPEND(pe,1,expectIntVar,susp_count);
  OZ_EXPECT_SUSPEND(pe,2,expectIntVar,susp_count);

  if (susp_count > 1)
    return pe.suspend();

  return pe.impose(new AddProp(OZ_in(0),
                               OZ_in(1),
                               OZ_in(2)));
}
OZ_BI_end
```

The variable `susp_count` is passed to the `OZ_EXPECT_SUSPEND` macros and if it is greater than 1 the function `fd_add_nestable()` is suspended. Otherwise the propagator is imposed.

1.4.3 Customizing `OZ_Expect`

The propagators implemented so far are imposed on single finite domain variables. The propagators will be resumed whenever an arbitrary element of a domain of its parameters is removed. But more elaborate propagators may have more demanding requirements concerning their resumption resp. parameter structure. Therefore the following frequently occurring requirements will be discussed in this section.

- Often it is *not* desirable to resume a propagator as soon as any arbitrary element is removed from the domain of one of its parameters. For instance, one might want to suspend resumption until a domain becomes a singleton domain.

- One wants to pass structured parameters to a propagator. In Section 1.5 a propagator will be implemented that expects a vector of finite domain variables.

The `expect` member functions are used to define new `expect` functions which specify the constraints for each parameter of a propagator which have to be entailed by the store to enable the imposition of the propagator. To conform with the macros `OZ_EXPECT` and `OZ_EXPECT_SUSPEND` the type of the resulting `expect` function has to be `OZ_expect_t (O::*)(OZ_Term)`, where `O` is either `OZ_Expect` or a class inheriting from it.

The new member function `expectIntVarSingl()` is implemented as member function of class `ExtendedExpect` inheriting from `OZ_Expect`. The definition of the member function `expectIntVarSingl()` which causes a propagator to be resumed when a variable is constrained to an integer, uses

```
OZ_Expect::expectIntVar(OZ_Term t,
                        OZ_FDPropState ps);
```

provided by the CPI . The second argument `ps` determines the event for resuming the propagator. For details on the values determining the resumption event see Section *Types*, *(The Mozart Constraint Extensions Reference)*.

The following code defines the class `ExtendedExpect` with the member function `expectIntVarSingl()`.

```
class ExtendedExpect : public OZ_Expect {
public:
  OZ_expect_t expectIntVarSingl(OZ_Term t) {
    return expectIntVar(t, fd_prop_singl);
  }
```

The definition of an `expect` function for vectors is similar. The CPI provides for the function

```
typedef OZ_expect_t (O::*OZ_ExpectMeth)(OZ_Term);
OZ_expect_t OZ_Expect::expectVector(OZ_Term v,
                                    OZ_ExpectMeth f);
```

which can be used to define a new instance of `expectVector` with the required signature `OZ_expect_t (O::*)(OZ_Term)`. The semantics of `expectVector` defines that `v` is a vector and all elements of the vector are constrained according to `f`, which is an `expect` function too.

Note that for a member function passed to `expectVector`, defined in a class inheriting from `OZ_Expect`, the cast `OZ_ExpectMeth` is necessary, since the type system of C/C++ cannot figure out by itself that the type of the function passed is admissible.

The following code is part of the definition of class `ExtendedExpect`.

```
private:
  OZ_expect_t _expectIntVarAny(OZ_Term t) {
    return expectIntVar(t, fd_prop_any);
  }
public:
  OZ_expect_t expectVectorIntVarAny(OZ_Term t) {
    return expectVector(t,
             (OZ_ExpectMeth) &_expectIntVarAny);
  }
  OZ_expect_t expectVectorIntVarSingl(OZ_Term t) {
    return expectVector(t,
             (OZ_ExpectMeth) &expectIntVarSingl);
  }
```

The implementation of the propagators discussed in the next sections assumes the existence of the class `ExtendedExpect`.

1.5 Using Vectors in Propagators

This section explains how propagators with vectors as parameters can be implemented by the CPI .

1.5.1 The *element* Constraint

The previous section explained techniques of how to handle propagator parameters which are vectors. This section implements the constraint $element(n, [d_1, \ldots, d_m], v)$, whose declarative semantics is defined by $d_n = v$.

All parameters of the *element* propagator are allowed to be finite domain variables resp. a vector of finite domain variables. We have the following propagation rules, which determine the operational semantics of the propagator for the constraint *element*.

Rule 1: $n := dom(n) \cap \{i \mid \exists j \in dom(v) : j \in dom(d_i)\}$

Rule 2: $v := dom(v) \cap \left(\bigcup_{i \in (dom(n) \cap \{1, \ldots, m\})} dom(d_i) \right)$

Rule 3: if $dom(n) = \{o\}$ then $d_o := dom(v)$

Note that $dom(x)$ denotes the current domain of x and $x := d$ denotes the update of $dom(x)$ with d.

The first rule states that the domain of n can only contain values i such that $dom(d_i)$ and $dom(v)$ share at least one value. The propagation rule (Rule 2) states that the domain of v cannot contain any value which does not occur in at least one d_i indexed by the values i of the domain of n, i.e. $i \in dom(n)$. The third rule says that as soon as n is a singleton containing o, the oth element of d is equal to v. The implementation of these rules is given in Section 1.5.5.

1.5.2 The Class Definition of `ElementProp`

The state of an instance of the class `ElementProp` contains a pointer to an array of `OZ_Term`s, namely `_d`. This is necessary, since it is not known beforehand how many elements are contained in the vector *d*. The size of the vector is stored in `_d_size`. Using a dynamic array in the state has some significant implications to the implementation of the member functions. The first function concerned is the constructor which has to allocate sufficient heap memory for the vector. The CPI provides the function `OZ_vectorSize()`, which computes the size of a vector passed as `OZ_Term`. This size is used to allocate an appropriately sized chunk of memory using the CPI function `OZ_hallocOzTerms()`. Finally, the vector as Oz data structure has to be converted to a C/C++ array. For convenience, the CPI provides the function `OZ_getOzTermVector()` which does this conversion. The following code gives the class definition described so far.

```cpp
class ElementProp : public OZ_Propagator {
private:
  static OZ_PropagatorProfile profile;
  OZ_Term _n, _v, * _d;
  int _d_size;
public:
  ElementProp(OZ_Term n, OZ_Term d, OZ_Term v)
    : _n(n), _v(v), _d_size (OZ_vectorSize(d))
  {
    _d =  OZ_hallocOzTerms(_d_size);
    OZ_getOzTermVector(d, _d);
  }

  virtual OZ_Return propagate(void);

  virtual size_t sizeOf(void) {
    return sizeof(ElementProp);
  }
  virtual OZ_PropagatorProfile *getProfile(void) const {
    return &profile;
  }
  virtual OZ_Term getParameters(void) const;
  virtual void gCollect(void);
  virtual void sClone(void);
};

OZ_PropagatorProfile ElementProp::profile;
```

The function `getParameters()` returns the arguments of the propagator in a list. Thereby, the vector *d* is represented in a sublist. The local C/C++ variable `list` is used to build up the list from the end of the vector. Therefore it is initialised as empty list and extended element-wise.

```cpp
OZ_Term ElementProp::getParameters(void) const {
```

```
    OZ_Term list = OZ_nil();

    for (int i = _d_size; i--; )
      list = OZ_cons(_d[i], list);

    return OZ_cons(_n,
                   OZ_cons(list,
                           OZ_cons(_v, OZ_nil()))));
  }
```

The member functions `gCollect()` and `sClone()` update the propagator's references
to the heap after the propagator has been copied by garbage collection or space cloning.
Updating the data members `_n` and `_v` is done by applying `OZ_gCollectTerm()` resp.
`OZ_sCloneTerm` to them. Updating the array `_d` requires to duplicate the array and then
to update all elements. This funtionality is provided by `OZ_gCollectAllocBlock()`
(`OZ_sCloneAllocBlock()`) for garbage collection (space cloning). Here comes the
code of those member functions.

```
  void ElementProp::gCollect(void) {
    OZ_gCollectTerm(_n);
    OZ_gCollectTerm(_v);
    _d = OZ_gCollectAllocBlock(_d_size, _d);
  }

  void ElementProp::sClone(void) {
    OZ_sCloneTerm(_n);
    OZ_sCloneTerm(_v);
    _d = OZ_sCloneAllocBlock(_d_size, _d);
  }
```

1.5.3 The Header Function

The implementation of the C/C++ function to impose the *element* propagator is straight-
forward with the techniques presented in Section 1.4. Note that this C/C++ function
treats empty vectors separately, since an empty list (resp. literal) is a valid vector, but
the *element* constraint is not defined on empty vectors. Therefore, the header function
is left via the member function **fail**() in case a vector of length 0 is detected.

```
  OZ_BI_define(fd_element, 3, 0)
  {
    OZ_EXPECTED_TYPE(OZ_EM_FD
                     ","OZ_EM_VECT OZ_EM_FD
                     ","OZ_EM_FD);

    ExtendedExpect pe;

    OZ_EXPECT(pe, 0, expectIntVar);
    OZ_EXPECT(pe, 1, expectVectorIntVarAny);
```

```
    OZ_EXPECT(pe, 2, expectIntVar);

    if (OZ_vectorSize(OZ_in(1)) == 0)
      return pe.fail();

    return pe.impose(new ElementProp(OZ_in(0),
                                     OZ_in(1),
                                     OZ_in(2)));
}
OZ_BI_end
```

The implementation uses the class `ExtendedExpect` (as explained in Section 1.4.3 since the imposition of this propagator requires to check if the second argument is a vector of finite domain variables and this functionality is not directly provided by the CPI class `OZ_Expect`.

1.5.4 Iterators Make Life Easier

The propagator for the *element* constraint operates on a vector, which is represented by an array in the state of the propagator. The member function `propagate()` has to apply certain functions, like `leave()` and `fail()`, to all elements of the array at once and not to individual elements. Therefore, it makes sense to define an *iterator* for such data structures.

The following code presents an iterator class for an `OZ_FDIntVar` array, which will be used by the member function `propagate()` of the *element* propagator.

```
class Iterator_OZ_FDIntVar {
private:
  int _l_size;
  OZ_FDIntVar * _l;
public:
  Iterator_OZ_FDIntVar(int s, OZ_FDIntVar * l)
    : _l_size(s), _l(l) { }

  OZ_Boolean leave(void) {
    OZ_Boolean vars_left = OZ_FALSE;
    for (int i = _l_size; i--; )
      vars_left |= _l[i].leave();
    return vars_left;
  }
  void fail(void) {
    for (int i = _l_size; i--; _l[i].fail());
  }
};
```

The iterator class provides the member functions `leave()` and `fail()` which call in turn the corresponding member functions on all elements of the array `l`. The function `leave()` returns 1 if there is at least one non-singleton domain left.

1.5.5 Implementing the Propagation Rules

The `propagate()` member function implements the propagation rules presented in Section 1.5.1 for the constraint $element(n, [d_1, \ldots, d_m], v)$.

The function `propagate()` defines local variables of type `OZ_FDIntVar`. Then it initializes the iterator object `D`. That avoids having to apply a member function to each element of `d` by hand if all elements have to be considered. The following **for**-loop initializes the elements of `d`.

The code coming after the implementation of the propagation rules (see below) checks if there is a non-singleton domain left, and if so it returns `OZ_SLEEP`. Otherwise the propagator is entailed and consequently returns `OZ_ENTAILED`. The label `failure` is provided because of the use of the macro `FailOnEmpty` (see Section 1.2.2.2) and the corresponding code applies **fail** to all variables of type `OZ_FDIntVar`.

```
OZ_Return ElementProp::propagate(void)
{
  OZ_FDIntVar n(_n), v(_v), d[_d_size];
  Iterator_OZ_FDIntVar D(_d_size, d);

  for (int i = _d_size; i--; )
    d[i].read(_d[i]);

  { /* propagation rule for n */
    OZ_FiniteDomain aux_n(fd_empty);

    for (int i = _d_size; i --; )
      if ((*(d[i]) & *v) != fd_empty)
        aux_n += (i + 1);

    FailOnEmpty(*n &= aux_n);
  }
  { /* propagation rule for v */
    OZ_FiniteDomain aux_v(fd_empty);

    for (int i = n->getMinElem();
         i != -1;
         i = n->getNextLargerElem(i))
      aux_v = aux_v | *(d[i - 1]);

    FailOnEmpty(*v &= aux_v);
  }
  { /* propagation rule for d[n] */
    if (n->getSize() == 1) {
      int o = n->getSingleElem();
      D.leave(); n.leave(); v.leave();
      return replaceBy(_v, _d[o - 1]);
    }
  }
```

```
    return (D.leave() | n.leave() | v.leave())
      ? OZ_SLEEP : OZ_ENTAILED;

failure:
    D.fail(); n.fail(); v.fail();
    return OZ_FAILED;
}
```

The propagation rules are implemented in the same order as they are presented in Section 1.5.1. That ensures that the values for i in rule 2 (page 19) are always in the index range of the vector `d`, since rule 1 (page 19) makes sure that only valid indices are contained in the domain of n. Note that the indices of vectors in Oz range over $1 \ldots n$ and the corresponding indices of C/C++ arrays over $0 \ldots n-1$.

Implementation of propagation rules   The implementation of the propagation rule 1 (page 19) starts with an initially empty auxiliary domain `aux_n`. It collects all integers i in the auxiliary domain, where the intersection of d_i and v is not empty. That is equivalent to finding at least one j being contained in d_i and v. The domain of n, i.e. `n`, is constrained by `aux_n`.

The second rule 2 (page 19) states that the domain of v cannot contain values that are not possible elements of the vector d. The implementation uses again an initially empty auxiliary domain `aux_v` and collects in a loop all elements of d_i in `aux_v` by iterating over all i being contained in n. The implementation of this rule closes with constraining `v` by `aux_v`.

The last rule, rule 3 (page 19), is only applied if n is determined, i.e. `n->getSize()` returns 1. Then it retrieves the value o, applies `leave()` to all variables of type `OZ_FDIntVar` and replaces the *element* propagator by the equality $v = d_o$ using the member function `replaceBy()` of class `OZ_Propagator` (see Section 1.3).

1.6   Connecting Finite Domain and Finite Set Constraints

The propagator in this section involves apart from finite domain constraints also finite set constraints. Its semantics is straightforward: it connects a domain variable D and a set variable S by projecting the changes of the constraints in both directions. Hereby, the finite domain variable designates an integer i (as usual) and the set variable designates a singleton set $\{e\}$ where $i = e$.

Propagation Rules   There are three propagation rules:

1. $\#S = 1$

2. $D \subseteq S$

3. $S \subseteq D$

The last two rules propagate the changes of the upper bound of the set constraint to the domain constraint and the other way around.

1.6.1 The Class Definition

The class definition does not have any particularities. It follows the scheme known from previous sections.

```
class ConnectProp : public OZ_Propagator {
private:
  static OZ_PropagatorProfile profile;
protected:
  OZ_Term _fs;
  OZ_Term _fd;
public:
  ConnectProp(OZ_Term fsvar, OZ_Term fdvar)
    : _fs(fsvar), _fd(fdvar)  {}

  virtual void gCollect(void) {
    OZ_gCollectTerm(_fd);
    OZ_gCollectTerm(_fs);
  }

  virtual void sClone(void) {
    OZ_sCloneTerm(_fd);
    OZ_sCloneTerm(_fs);
  }

  virtual size_t sizeOf(void) {
    return sizeof(ConnectProp);
  }

  virtual OZ_Term getParameters(void) const {
    return OZ_cons(_fs, (OZ_cons(_fd, OZ_nil())));
  }

  virtual OZ_PropagatorProfile *getProfile(void) const {
    return &profile;
  }

  virtual OZ_Return propagate();
};
```

Note that set variables are handled the same way as domain variables.

1.6.2 The Propagation Function

The implementation of the propagation function starts with retrieving the constrained variables from the constraint store using the constructors of the classes `OZ_FDIntVar` and `OZ_FSetVar`. The class `OZ_FSetVar` provides for the same member functions as `OZ_FDIntVar` such that handling set variables does not differ from handling domain variables.

Propagation The propagation starts with the first rule. it uses the member function `OZ_FSetConstraint::putcard(int, int)` to impose the cardinality constraint upon *S*. The second rule implemented by removing all elements from *D* that are definitely not in *S* (see Section *Reflection Member Functions*, *(The Mozart Constraint Extensions Reference)* for details on `OZ_FSetConstraint::getNotInSet()`). The last propagation rule uses the operator `OZ_FSetConstraint::operator <=` for $S \subseteq D$. The constructor `OZ_FSetConstraint` is used to convert the `OZ_Finite Domain` appropriately. Note that imposing constraints on *D* resp. *S* are guarded by `FailOnEmpty` resp. `FailOnInvalid` to catch failures.

```
OZ_Return ConnectProp::propagate() {

printf("ConnectProp::propagate\n");

  OZ_FDIntVar fd(_fd);
  OZ_FSetVar  fs(_fs);

  // 1st propagation rule
  fs->putCard(1, 1);

  // 2nd propagation rule
  FailOnEmpty(*fd -= fs->getNotInSet());

  // 3rd propagation rule
  FailOnInvalid(*fs <= OZ_FSetConstraint(*fd));

  return (fd.leave() | fs.leave()) ? OZ_SLEEP : OZ_ENTAILED;

failure:
  fd.fail(); fs.fail();
  return OZ_FAILED;
}
```

The macro `FailOnInvalid` is define as

```
#define FailOnInvalid(X) if(!(X)) goto failure;
```

since finite set operator return `OZ_FALSE` in case an inconsistency occured.

The propagator closes with calling `leave()` for both variables and returning `OZ_SLEEP` resp. `OZ_ENTAILED` depending on whether not all variables denote values or they do.

1.6.3 The Header Function and Connecting to the Native Functor Interface

The header function uses `OZ_Expect::expectFSetVar` to check for a set variable.

```
OZ_BI_define(connect, 2, 0)
{
  OZ_EXPECTED_TYPE(OZ_EM_FSET","OZ_EM_FD);
```

```
      OZ_Expect pe;
      OZ_EXPECT(pe, 0, expectFSetVar);
      OZ_EXPECT(pe, 1, expectIntVar);

      return pe.impose(new ConnectProp(OZ_in(0), OZ_in(1)));
    }
    OZ_BI_end


    OZ_PropagatorProfile ConnectProp::profile;
```

The predefined macro `OZ_EM_FSET` is used to produce an appropriate error message in case an type exception has to be risen.

The C part of the native functor interface is given below.

```
    OZ_C_proc_interface *oz_init_module(void)
    {
      static OZ_C_proc_interface i_table[] = {
        {"connect", 2, 0, connect},
        {0, 0, 0, 0}
      };

      return i_table;
    }
```

1.6.4 Testing the Propagator

To make the propagator available on Oz side feed the following code:

```
    declare
    Connect = {{New Module.manager init}
               link(url: 'sync.so{native}' $)}.connect
    {Wait Connect}
    {Show Connect}
```

The variable `Connect` refers to the propagator. By feeding the code below line by line one can observe (e.g. using the Browser *"The Oz Browser"*), how the propagator works.

```
    declare S = {FS.var.decl}
           I = {FD.decl}
                      % S =                    I =
    {Connect S I}    % {{}..{0#134217726}}#1    {0#134217726}
    {FS.exclude 2 S} % {{}..{0#1 3#134217726}}#1  {0#1 3#34217726}
    I :: 1#100       % {{}..{1 3#100}}#1         {1 3#100}
    {FS.exclude 1 S} % {{}..{3#100}}#1           {3#100}
    I <: 4           % {3}#1                     3
```

The comments at the end of each line indicate the constraints after feeding that line.

1.7   Advanced Topics

This section discusses issues of practical relevance not covered in this manual so far.
It explains implementation techniques rather than giving ready-to-use code.

1.7.1   Detecting Equal Variables in a Vector

A feature of the finite domain constraint system of Oz is that it is able to exploit equal-
ity between variables. For example, one can simplify linear equations in case equal
variables are detected. Let us regard the equation $2u + v + 7w - 2x + y = 0$. Imposing
the equality constraints $u = x$ and $v = y$ allows to simplify the equation to $7w + 2y = 0$.
This simplification offers the advantage that the propagator becomes computationally
less complex resulting in a better execution performance.

The CPI provides the function

OZ_findEqualVars   `int * OZ_findEqualVars(int sz, OZ_Term * v)`

to detect equal variables in an `OZ_Term` array. It expects `v` to be an array with `sz`
elements. Assume the application

```
int * pa = OZ_findEqualVars(arr_sz, x);
```

where `pa` is called the position array. The array `x` is scanned with ascending index
starting from 0 to determine the values of `pa`. If `x[i]` denotes a variable and this
variable occurs the first time, the value of `pa[i]` is `i`. In case the variable occurs not
the first time, `pa[i]` contains the index of the first occurrence. If `x[i]` denotes an
integer, `pa[i]` contains -1.

As an example, consider the constraint $2a + 3b - 4c - 5d + 4e + 8 = 0$ where at runtime
the constraint $c = e \land d = 2$ is imposed. The result of the equal variable detection is as
follows.

i	0	1	2	3	4
x[i]	a	b	c	d	e
pa[i]	0	1	2	-1	2

The state of the propagator can now be updated to represent the equivalent constraint
$2a + 3b - 2 = 0$. Thus, this simplification avoids tedious handling of equal variables
in the propagation algorithm and it improves memory consumption and runtime be-
haviour.

mayBeEqualVars   To avoid unnecessary calls of `OZ_findEqualVars()`, this func-
tion is intended to be used in conjunction with the member function `mayBeEqualVars()`
of class `OZ_Propagator` (see also Section *Provided Member Functions, (The Mozart
Constraint Extensions Reference)*). In case an equality constraint has been imposed
on at least one variable occurring in the propagator's parameters, `mayBeEqualVars()`
returns 1.

Note that the function `OZ_findEqualVars()` returns a pointer to a static array, i.e.
another application of this function will override the previous values.

1.7.2 Avoiding Redundant Copying

In Section 1.5.2 we learned that data structures referenced by the state of a propagator have to be copied whenever the Oz runtime system calls either the member function `gCollect()` or `sClone()`. But constant data structures, i.e. data structures which do not change during the propagator's lifetime, need only to be duplicated in case of a garbage collection. Otherwise it is sufficient to have a reference to such a constant data structure. Thus it is useful to use a reference counting technique to keep track of the number of references to the constant data structure, so that the destructor of the propagator can dispose the data structure when there is no reference left.

That is one reason why there are distinct member functions for garbage collection and space cloning. Garbage collection requires a fresh copy of constant data structures while space cloning requires only a reference and a reference counting technique is applicable.

The code presented in this section defines the class `ConstDataHdl` which can be used to avoid redundant copying of constant data structures by approptiate actions in `gCollect()` and `sClone()`. The class `ConstDataHdl` implements a reference counting scheme and holds in its state, apart from the actual constant data structure, the reference counter `_refCount` and the forward reference `_newLoc`. In our example the constant data structure is the string `"Constant data"`.

The constructor of `ConstDataHdl` creates the constant data structure and initialises the reference counting mechanism. The operator `new` is redefined to allocate instances of `ConstDataHdl` on the heap. The operator `delete` decrements the reference counter and deallocates the instance of `ConstDataHdl` from the heap if there is no reference left. The member function `getRef()` is to be used if a new reference to an instance of `ConstDataHdl` is needed (`sClone()`). It increments `_refCount` and returns the self-reference `this`. The member function `copy()` is to be used if the constant data structure has to be duplicated which is the case in `gCollect()`.

```
class ConstDataHdl {
private:
  char _constData[100];

  int _refCount;
  ConstDataHdl * _newLoc;
public:
  ConstDataHdl(char * str)
    : _refCount(1), _newLoc(NULL) {
      strcpy(_constData, str);
  }
  static void * operator new (size_t sz) {
    return OZ_hallocChars(sz);
  }
  static void operator delete (void * p) {
    if (0 == --((ConstDataHdl *) p)->_refCount)
      OZ_hfreeChars((char *) p,sizeof(ConstDataHdl));
  }
  ConstDataHdl * getRef(void) {
```

```
              _refCount += 1;
              return this;
          }
          ConstDataHdl * copy (void) {
            if (_newLoc)
              _newLoc->getRef();
            else
              _newLoc = new ConstDataHdl(_constData);
            return _newLoc;
          }
      };
```

At its first invocation the member function `copy()` duplicates the instance it is called from, sets the forward reference `newLoc` to the location of the duplicate, and returns the reference to the duplicate. All subsequent invocations only increment the reference counter of the duplicate and return a reference to the duplicate.

To use the presented reference counting scheme in a propagator add to ...

... the class definition of the propagator:

```
      ConstDataHdl * _constData;
```

... the constructor definition of the propagator:

```
      _constData = new ConstDataHdl("Constant data");
```

... the destructor definition of the propagator:

```
      delete _constData;
```

... the definition of the member function `gCollect()`:

```
      _constData = _constData->copy();
```

... the definition of the member function `sClone()()`:

```
      _constData = _constData->getRef();
```

The presented class definition of `ConstDataHdl` can be adopted by redefining the embedded data structure `ConstDataHdl::_constData` appropriately.

1.7.3 Reified Constraints

This section sketches the implementation of reified constraints (see the section on reified constraints in Chapter *Reified Constraints, (Finite Domain Constraint Programming in Oz. A Tutorial.)*) and goes into more details concerning the particularities of the class `OZ_FDIntVar`.

The idea of reification is as follows: A $0/1$-variable R is associated with a constraint C. The variable R is called *control variable*. As long as the domain of R is not constrained to a singleton domain, the constraint C checks if the constraint store entails or disentails C. If so, the variable R is constrained to 1 or 0, respectively. Otherwise, if R is constrained to 0 or 1 then the constraint C or $\neg C$, respectively, is imposed to the store.

The implementation of a reified constraint is explained for $(x \leq y) \leftrightarrow r$, which will be implemented by the class `ReifiedLessEqProp`. We assume that the constraints $x \leq y$ and $x > y$ are implemented by the classes `LessEqProp` resp. `GreaterProp`. This section focuses on implementing `ReifiedLessEqProp::propagate()`.

There are basically two cases to be regarded. The first case is that the domain of the control variable is an integer. Then $(x \leq y) \leftrightarrow r$ has to be replaced either by $x \leq y$ or by $x > y$. The technique to replace a propagator by another one is explained in Section 1.3.

Encapsulated Constraint Propagation   If the control variable is still a $0/1$ variable, the reified propagator checks if the constraint $x \leq y$ is entailed resp. disentailed by the store. For this, the propagator has to perform a constraint propagation such that the propagation results are only locally visible inside the propagator and not written to the store. This is called *encapsulated constraint propagation*. Additionally, the reified propagator checks if the constraints produced by encapsulated propagation, so-called *encapsulated constraints*, are subsumed by the constraint store. If so the control variable is constrained to 1. If the encapsulated constraints are inconsistent, the control variable is constrained to 0. Otherwise the control variable is left untouched.

The member function `readEncap`   Instances of class `OZ_FDIntVar` are usually initialised by the member function `read()` or the constructor `OZ_FDIntVar(OZ_Term)` with the intention to make amplified constraints visible to the store. To obtain an instance of `OZ_FDIntVar()` providing encapsulated constraint propagation, the function `readEncap()` has to be used instead. Such an instance is used in the same way as in the non-encapsulated case.

The code below implements member function `propagate()` of class `ReifiedLessEq`. It is implemented in such a way that it utilises encapsulated propagation [2].

```
OZ_Return ReifiedLessEqProp::propagate()
{
  OZ_FDIntVar r(_r);

  if(*r == fd_singl) {
    r.leave();
    return replaceBy((r->getSingleElem() == 1)
                      ? new LessEqProp(_x, _y)
                      : new GreaterProp(_x, _y));
  }

  OZ_FDIntVar x, y;
```

---

[2]Of course, an alternative would have been to reason over the bounds of the domains.

```
      x.readEncap(_x); y.readEncap(_y);
      int r_val = 0;

      // entailed by store?
      if (x->getMaxElem() <= y->getMinElem()) {
        r_val = 1;
        goto quit;
      }

      if (0 == (*x <= y->getMaxElem())) goto quit;
      if (0 == (*y >= x->getMinElem())) goto quit;

      r.leave(); x.leave(); y.leave();
      return OZ_SLEEP;

quit:
      if(0 == (*r &= r_val)) {
        r.fail(); x.fail(); y.fail();
        return OZ_FAILED;
      }

      r.leave(); x.leave(); y.leave();
      return OZ_ENTAILED;
    }
```

The implementation checks first whether the control variable `r` denotes a singleton. If so, the reified propagator is replaced by an appropriate propagator depending on the value of `r`.

Otherwise the code proceeds with defining the variables `x` and `y` as instances of class `OZ_FDIntVar`. Initialisation of `x` and `y` with `readEncap()` ensures encapsulated constraint propagation. Next it is checked if $x \leq y$ is entailed by the store, which is the case if $\overline{x} \leq \underline{y}$ is true [3]. If so, `r_val` is set to 1 and the code branches to label `quit`. Then the propagation rules are implemented. They are $x \leq \overline{y}$ and $y \geq \underline{x}$. In case an inconsistency is detected, the code branches to label `quit` and the value of `r_val` is left at 0. Finally, the function `propagate()` returns `OZ_SLEEP` to the runtime system.

The code at label `quit` constrains `r` to the value of `r_val` and in case of an inconsistency it returns `OZ_FAILED`. Otherwise the propagator is left by returning `OZ_ENTAILED`.

---

[3]Note that $\underline{x}$ ($\overline{x}$) denotes the smallest (largest) integer of the current domain of x

Building Constraint Systems from Scratch

2.1 The Generic Part of the CPI

2.1.1 The Model of a Generic Constraint Solver

This section describes how to implement constraint systems from scratch. First we will explain the underlying concepts in an informal way and try to draw a big picture of the implementation.

Constraint propagation takes place in a *computation space* which consists of the *constraint store* and propagators associated with the constraint store. The constraint store holds variables that either refer to values (i.e., are bound resp. determined) or are unbound. But there may already be some information about the value an unbound variable will later refer to. For example, it might be already known that a variable refers to an integer. We say the variable is *constrained*, here by a finite domain constraint. This information is stored right at the variable. To provide a generic scheme to associate self-defined constraints with a variable in the constraint store, such a variable has a pointer of type `(OZ_Ct *)`, pointing to a constraint instance of the self-defined constraint system. That is done by defining new constraints as subclasses of `OZ_Ct`.

The main part of a propagator is its propagation routine. This routine fulfills mainly three tasks. First it retrieves the constraint from the constraint store. The class `OZ_CtVar` provides a generic interface for that task. Then the propagation algorithm generates new projections on the retrieved constraints. Finally the new constraints are written back to the constraint store. Usually a propagator parameter is shared between more than one propagator. Modifying a constraint in the constraint store may enable another propagators to generate new projections, hence when writing constraints to the store, propagators sharing parameters have to be notified. This is done by the appropriate member functions of `OZ_CtVar` but to decide what propagators to notify, the propagator has to memorize the constraints present in the constraint store before the propagation algorithm modified the store. The class `OZ_CtProfile` serves that purpose by providing a generic interface (used in `OZ_CtVar`) to store characteristic information of a constraint sufficient to derive which propagators have to be notified.

The notification of propagators is realized by wake-up lists associated with the constrained variable. Depending on the kind of constraint system there will be different

events a propagator may want to be notified upon. For each event there is a wake-up list. The wake-up events of a constraint system are determined by an object of a subclass of `OZ_CtDefinition`. Hence, upon creation of a new constrained variable a reference of type `OZ_CtDefinition *` has to be passed to `OZ_mkCtVariable()` which takes care of creating variables with generic constraints.

The following sections explain in detail how a constraint system can be implemented using the provided abstractions briefly mentioned in this section.

2.1.2   Overview over Generic Part of the CPI

The CPI provides abstractions to implement constraint systems from scratch. Five classes provide the required functionality. They allow to implement new constraint system at a high level of abstraction without sacrificing efficiency (e.g., it is straightforward to take advantage of wake-up lists for distinct wake-up events [lower bound changed, upper bound changed etc.]).

This part of the CPI is based on the principles developed by [5].

The following classes are provided:

OZ_CtDefinition

The class `OZ_CtDefinition` serves as an identifier for a particular constraint system and defines certain parameters for that constraint system, as for example the number of wake-up lists. See Section *The class* `OZ_CtDefinition` , *(The Mozart Constraint Extensions Reference)* for details.

OZ_Ct

The class `OZ_Ct` represents the actual constraint attached to a constrained variable. See Section *The class* `OZ_Ct` , *(The Mozart Constraint Extensions Reference)* for details.

OZ_CtVar

The class `OZ_CtVar` provides access to a constrained variable in the constraint store. Amongst other things, it provides the following services:

- handling of local and global variables transparently (trailing).
- Making the actual constraints in the store accessible from within a propagator to allow to manipulate them.
- Triggering the scanning of appropriate wake-up lists (using `OZ_CtProfile`).

See Section *The class* `OZ_CtVar` , *(The Mozart Constraint Extensions Reference)* for details.

OZ_CtProfile

The class `OZ_CtProfile` stores characteristic parameters of a constraint (called its *profile*) to determine the wake-up list(s) to be scanned. Typically, this happens when a propagator is left. See Section *The class* `OZ_CtProfile` , *(The Mozart Constraint Extensions Reference)* for details.

`OZ_CtWakeUp`

> An instance of the class `OZ_CtWakeUp` controls which wake-up lists have to be scanned and is produced by comparing the current state of a constraint and a previously taken constraint profile. See Section *The class* `OZ_CtWakeUp` , *(The Mozart Constraint Extensions Reference)* for details.

> Further, there is a function `OZ_mkCtVariable()` that allows to create a new constrained variable according to a given definition and a given constraint. Additionally, the class `OZ_Expect` provides the member function `OZ_Expect::expectGenCtVar()` to handle constrained variables appropriately.

> To demonstrate the usage of this part of the CPI, constraints over real-intervals are implemented in Section 2.2.

2.2   A Casestudy: Real Interval Constraints

2.2.1   An Implementation

> In real-interval constraints [1] a variable denotes a real number r. The constraint approximates r by a lower bound and a upper bound, i.e. $l \leq r \leq u$. The bounds are represented as floating point numbers. A floating point number itself is an approximation of a real number due to the limited precision of its machine representation. The implementation of real-interval constraints has to take care to not prune valid solutions by computing the bounds too tight. This has to be avoided by controlling the direction of rounding of the floating point operations.

> The *width* of a real-interval is the difference between its upper bound and its lower bound. If the width of a real-interval constraint is less than or equal to a given *precision* then the constraint is regarded a value.

> The goal of this section is to give an overview of how the CPI classes interact with each other and not to describe the implementation of a complete constraint system in too much detail. The definition of all required classes is only sketched and additionally, a simple propagator is implemented.

> Note that floating point numbers have the type `ri_float` which is compatible with float numbers used by the Oz runtime system. Such float numbers may range from `RI_FLOAT_MIN` to `RI_FLOAT_MAX`.

2.2.1.1   A Definition Class for Real-Interval Constraints

> The class `RIDefinition` is derived from the CPI class `OZ_CtDefinition`. It gathers all information needed to handle real-interval constraints properly by the runtime system. It allows the runtime system to distinguish real-interval constraint from other constraints by calling the member function `getKind()`. Note that `_kind` is static and to obtain a unique identifier the function `OZ_getUniqueId()` is recommended to be used. For testing Oz values to be compatible with real-intervals `isValidValue()`, is to be defined appropriately.

> Further, `RIDefinition` allows the runtime system to determine the number of possible events causing suspending computation to be woken up. There are two possible

events for which a suspending computation may want to be notified: the lower bound is increased or the upper bound is decreased (or both). Therefore, two wake-up lists are used (see `getNoOfWakeUpLists()`).

```cpp
class RIDefinition : public OZ_CtDefinition {
private:
  static int _kind;

public:
  virtual int getKind(void) { return _kind; }
  virtual char * getName(void) { return "real interval"; }
  virtual int getNoOfWakeUpLists(void) { return 2; }
  virtual char ** getNamesOfWakeUpLists(void) {
    static char * names[2] = {"lower", "upper"};
    return names;
  }
  virtual OZ_Ct * leastConstraint(void) {
    return RI::leastConstraint();
  }
  virtual OZ_Boolean isValidValue(OZ_Term f) {
    return RI::isValidValue(f);
  }
};

int RIDefinition::_kind = OZ_getUniqueId();
```

The function `leastConstraint()` is required to enable the runtime system to constrain a variable to a real-interval with greatest possible width, i.e., ranging from `RI_FLOAT_MIN` to `RI_FLOAT_MAX`. For example this is necessary when nested variables are to be constrained.

2.2.1.2 Determining Wake-up Events for Real-Interval Constraints

Instances of classes derived from `OZ_CtWakeUp` indicate to the runtime system which wake-up event occurred. Therefore, member functions to initialize an `RIWakeUp` instance according to a possible event are defined. They will be used to determine the wake-up event of a propagator upon a certain parameter when imposing a propagator(see Section 2.2.1.5).

```cpp
class RIWakeUp : public OZ_CtWakeUp {
public:
  static OZ_CtWakeUp wakeupMin(void) {
    OZ_CtWakeUp ri_wakeup_min;
    ri_wakeup_min.init();
    ri_wakeup_min.setWakeUp(0);
    return ri_wakeup_min;
  }
  static OZ_CtWakeUp wakeupMax(void) {
    OZ_CtWakeUp ri_wakeup_max;
```

```
      ri_wakeup_max.init();
      ri_wakeup_max.setWakeUp(1);
      return ri_wakeup_max;
    }
    static OZ_CtWakeUp wakeupMinMax(void) {
      OZ_CtWakeUp ri_wakeup_minmax;
      ri_wakeup_minmax.init();
      ri_wakeup_minmax.setWakeUp(0);
      ri_wakeup_minmax.setWakeUp(1);
      return ri_wakeup_minmax;
    }
};
```

An instance of `RIWakeUp` is computed from an instance of `RI` and a profile (stored before the constraint has be modified) by

```
OZ_CtWakeUp RI::getWakeUpDescriptor(OZ_CtProfile * p)
```

(see Section 2.2.1.3). The definition of the profile class `RIProfile` for real-intervals is given below.

```
class RIProfile : public OZ_CtProfile {
private:
  ri_float _l, _u;

public:
  RIProfile(void) {}
  virtual void init(OZ_Ct * c) {
    RI * ri = (RI *) c;
    _l = ri->_l;
    _u = ri->_u;
  }
};
```

The function `RIProfile::init(OZ_Ct * c)` takes a snapshot of `c` to enable the detection of the abovementioned wake-up events, i.e., modified lower/upper bound resp. both.

2.2.1.3 The Actual Real-Interval Constraint

The actual real-interval constraint is represented by instances of the class `RI`. It stores the upper and lower bound, to approximate a real number. Apart from constructors and initialization functions a couple of general functions are defined.

```
class RI : public OZ_Ct {
private:
  ri_float _l, _u;

public:
```

```
RI(void) {}
RI(ri_float l, ri_float u) : _l(l), _u(u) {}

void init(OZ_Term t) { _l = _u = OZ_floatToC(t);   }

ri_float getWidth(void) { return _u - _l; }

static OZ_Ct * leastConstraint(void) {
  static RI ri(RI_FLOAT_MIN, RI_FLOAT_MAX);
  return &ri;
}

static OZ_Boolean isValidValue(OZ_Term f) {
  return OZ_isFloat(f);
}

OZ_Boolean isTouched(RIProfile rip) {
  return (rip._l < _l) || (rip._u > _u);
}
...
```

The member functions `leastConstraint()` and `isValidValue()` are used by class `RIDefinition` and their definition is self-explanatory. Note that `ri_float` values have to be compatible with Oz float such that `OZ_float()` is used for testing compatibility.

Most of the definitions of virtual member functions and operators used for implementing constraint propagation are self-explanatory. Note that `copy()` uses the `new` operator provided by `OZ_Ct` and the constructor `RI(ri_float,ri_float)`. The function `isValue()` assumes a global variable `ri_float ri_precision;` that holds the current precision.

```
...
virtual char * toString(int);
virtual size_t sizeOf(void);

virtual OZ_Ct * copy(void) {
  RI * ri = new (sizeof(ri_float)) RI(_l, _u);
  return ri;
}

virtual OZ_Boolean isValue(void) {
  return (getWidth() < ri_precision);
}

virtual OZ_Term toValue(void) {
  double val = (_u + _l) / 2.0;
  return OZ_float(val);
}
```

```
      virtual OZ_Boolean isValid(void) {
        return _l <= _u;
      }
      ...
```

The function `getWakeUpDescriptor()` computes from the current state of the con-
straint and a given constraint profile `p` a wake-up descriptor. Therefore, it creates an
empty one and sets the appropriate events successively. Finally it returns the descriptor.

```
      ...
      virtual RIProfile * getProfile(void) {
        static RIProfile rip;
        rip.init(this);
        return &rip;
      }

      virtual
      OZ_CtWakeUp getWakeUpDescriptor(OZ_CtProfile * p) {
        OZ_CtWakeUp d;
        d.init();

        RIProfile * rip = (RIProfile *) p;
        if (_l > rip->_l) d.setWakeUp(0);
        if (_u < rip->_u) d.setWakeUp(1);

        return d;
      }
      ...
```

The function `isWeakerThan()` simply compares the widths of two real-interval con-
straints to detect whether the constraint `*this` is subsumed by `*r`. This makes sense
since `*this` represents never values not represented by `*r` which is ensured by the
runtime system.

The unification routine for two real-interval constraints computes the intersection of
the values approximated by `*this` and `*r`. The result is stored in a static variable and
eventually a pointer to this variable is returned.

```
      ...
      virtual OZ_Boolean isWeakerThan(OZ_Ct * r) {
        RI * ri = (RI *) r;
        return (ri->getWidth() < getWidth());
      }

      virtual OZ_Ct * unify(OZ_Ct * r) {
        RI * x = this, * y = (RI *) r;
        static RI z;

        z._l = max(x->_l, y->_l);
```

```
      z._u = min(x->_u, y->_u);

      return &z;
    }

    virtual OZ_Boolean unify(OZ_Term rvt) {
      if (isValidValue(rvt)) {
        double rv = OZ_floatToC(rvt);

        return (_l <= rv) && (rv <= _u);
      }
      return 0;
    }
    ...
```

The unification routine of a real-interval constraint and a value checks if the value is compatible with float numbers and then, if the value is contained in the set of values represented by the constraint. Note that this function indicates only if a unification is successful and does not update the constraint.

The operators for constraint propagation are straight-forward. They return the width of the computed constraint. In case the width is less zero, the constraint is inconsistent and thus can be easily tested.

```
    ...
    ri_float operator <= (ri_float f) {
      _u = min(_u, f);
      return getWidth();
    }

    ri_float operator >= (ri_float f) {
      _l = max(_l, f);
      return getWidth();
    }

    ri_float lowerBound(void) { return _l; }
    ri_float upperBound(void) { return _u; }
    ...
}; // class RI
```

The functions `lowerBound()` and `upperBound()` provide access to the lower resp. upper bound of the constraint.

2.2.1.4 Accessing the Constraint Store

The class `RIVar` is defined to provide access to a real-interval variable in the constraint store. The class `RIVar` is derived from `OZ_CtVar`. The private and protected part of the class definition of `RIVar` is the implementation of the principle described in Section *The class* `OZ_CtVar` , *(The Mozart Constraint Extensions Reference)* for real-interval constraints.

```cpp
class RIVar : public OZ_CtVar {
private:

  RI * _ref;
  RI _copy, _encap;

  RIProfile _rip;

protected:

  virtual void ctSetValue(OZ_Term t)
  {
    _copy.init(t);
    _ref = &_copy;
  }

  virtual OZ_Ct * ctRefConstraint(OZ_Ct * c)
  {
    return _ref = (RI *) c;
  }

  virtual OZ_Ct * ctSaveConstraint(OZ_Ct * c)
  {
    _copy = *(RI *) c;
    return &_copy;
  }

  virtual OZ_Ct * ctSaveEncapConstraint(OZ_Ct * c)
  {
    _encap = *(RI *) c;
    return &_encap;
  }

  virtual void ctRestoreConstraint(void)
  {
    *_ref = _copy;
  }

  virtual void ctSetConstraintProfile(void)
  {
    _rip = *_ref->getProfile();
  }

  virtual OZ_CtProfile * ctGetConstraintProfile(void)
  {
    return &_rip;
  }

  virtual OZ_Ct * ctGetConstraint(void)
```

```
    {
      return _ref;
    }

public:

    RIVar(void) : OZ_CtVar() { }

    RIVar(OZ_Term t) : OZ_CtVar() { read(t); }

    virtual OZ_Boolean isTouched(void) const
    {
      return _ref->isTouched(_rip);
    }

    RI &operator * (void) { return *_ref; }
    RI * operator -> (void) { return _ref; }

};
```

The public part of the class definition is self-explanatory. It provides for constructors, the function `isTouched()` to enable the CPI to detect if a parameter has been changed, and operators to provide direct access to the real-interval constraints.

2.2.1.5 Extending `OZ_Expect`

Propagators are imposed on their parameters by foreign functions which are invoked by the Oz runtime system. Such foreign functions use the CPI class `OZ_Expect` to check that the parameters are appropriately kinded (resp. constrained) or represent compatible values. The class `OZ_Expect` provides the member function

```
OZ_expect_t OZ_Expect::expectGenCtVar(OZ_Term t,
                                      OZ_CtDefinition * d,
                                      OZ_CtWakeUp w);
```

to define appropriate expect-functions, e.g., for real-interval constraints. The customized class defines member functions that check for real-intervals and determine the wake-up event. To do that, the static members functions of `RIWakeUp` (see Section 2.2.1.2) are used and the global variable `RIDefinition ri_definition` is assumed.

```
class RIExpect : public OZ_Expect {
public:
  OZ_expect_t expectRIVarMin(OZ_Term t) {
    return expectGenCtVar(t, ri_definition,
                          RIWakeUp::wakeupMin());
  }
  OZ_expect_t expectRIVarMax(OZ_Term t) {
    return expectGenCtVar(t, ri_definition,
```

```
                                              RIWakeUp::wakeupMax());
  }
  OZ_expect_t expectRIVarMinMax(OZ_Term t) {
    return expectGenCtVar(t, ri_definition,
                          RIWakeUp::wakeupMinMax());
  }
  ...
};
```

The class `RIEexpect` can now be used to define foreign functions that impose propagators on their parameters.

```
OZ_BI_define(ri_lessEq, 2, 0)
{
  OZ_EXPECTED_TYPE("real interval, real interval");

  RIExpect pe;

  OZ_EXPECT(pe, 0, expectRIVarMinMax);
  OZ_EXPECT(pe, 1, expectRIVarMinMax);

  return pe.impose(new RILessEq(OZ_args[0],
                                OZ_args[1]));
}
OZ_BI_end
```

The propagator class `RILessEq` is partly defined next.

2.2.1.6 A Simple Propagator

The description of the implementation of real-interval constraints is closed with the discussion of the propagation function of a simple propagator, namely a propagator for the constraint $x \leq y$. The rest of the class definition of that propagator is similar to other propagators and therefore omitted here.

```
OZ_Return RILessEq::propagate(void)
{
  RIVar x(_x), y(_y);

  // step (1)
  if (x->upperBound() <= y->lowerBound()) {
    x.leave(); y.leave();
    return OZ_ENTAILED;
  }

  // step (2)
  if((*x <= y->upperBound()) < 0.0)
    goto failure;
```

```
                      // step (3)
                      if((*y >= x->lowerBound()) < 0.0)
                        goto failure;

                      return (x.leave() | y.leave())
                        ? OZ_SLEEP : OZ_ENTAILED;

                  failure:
                      x.fail(); y.fail();
                      return OZ_FAILED;
                  }
```

Assume that the propagator stores in its state references to its parameters on the Oz heap by the members `OZ_Term _x, _y;`. The function `propagate()` obtains access to the constraint store by declaring two instances of class `RIVar` and passing the Oz terms `_x` and `_y` as arguments.

The propagation proceeds in three steps.

1. Test if the constraint $x \leq y$ is subsumed by the constraint store, i.e., $\overline{x} \leq \underline{y}$.

2. Constrain the upper bound of x: $x \leq \overline{y}$.

   That is implemented by `ri_float RI::operator <= (ri_float)`.

3. Constrain the lower bound of y: $y \geq \underline{x}$.

   That is implemented by `ri_float RI::operator >= (ri_float)`.

As said in Section 2.2.1.3 these operators return the width of the computed constraint. A width less than 0 indicates that the constraint has become inconsistent and propagation branches to label `failure`.

The function `OZ_CtVar::leave()` returns `OZ_True` if the constraint does not denotes a value. This is used to detect whether the propagator has to be rerun (indicated by `OZ_SLEEP`) or not (indicated by `OZ_ENTAILED`).

The return value `OZ_FAILED` informs the runtime system that the constraint is inconsistent with the constraint store.

2.2.2 The Reference of the Implemented Real-Interval Constraint Solver

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

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

or by

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

---

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

as part of a functor definition.

RI.inf

An implementation-dependent float value that denotes the smallest possible float number. It is -1.79769×10^{308}.

RI.sup

An implementation-dependent float value that denotes the smallest possible float number. It is 1.79769×10^{308}.

{RI.setPrec $+$F**}**

Sets the precision of the real-interval constraints to F.

{RI.getLowerBound $+$RI ?F**}**

Returns the lower bound of RI in F.

{RI.getUpperBound $+$RI ?F**}**

Returns the upper bound of RI in F.

{RI.getWidth $+$RI ?F**}**

Returns the width of RI in F.

{RI.var.decl ?RI**}**

Constrains RI to a real-interval constraint with the lower bound to be RI.inf and the upper bound to be RI.sup.

{RI.var.bounds $+$L $+$U ?RI**}**

Constrains RI to a real-interval constraint with the lower bound to be L and the upper bound to be U.

{RI.lessEq $X $Y**}**

Imposes the constraint $\mathtt{X} \leq \mathtt{Y}$.

{RI.greater $X $Y**}**

Imposes the constraint $\mathtt{X} > \mathtt{Y}$.

{RI.intBounds $RI $D**}**

Imposes the constraint $\lceil \underline{\mathtt{RI}} \rceil = \underline{\mathtt{D}} \wedge \lfloor \overline{\mathtt{RI}} \rfloor = \overline{\mathtt{D}}$.

{RI.times $X $Y $Z**}**

Imposes the constraint $\mathtt{X} \times \mathtt{Y} = \mathtt{Z}$.

{RI.plus $X $Y $Z**}**

Imposes the constraint $\mathtt{X} + \mathtt{Y} = \mathtt{Z}$.

{RI.distribute $*$RI**}**

Creates a choice-point for $\mathrm{RI} \leq m$ and $\mathrm{RI} > m$ where $m = \underline{\mathtt{RI}} + (\overline{\mathtt{RI}} - \underline{\mathtt{RI}})/2$.

3

Employing Linear Programming Solvers

3.1 Introduction

The introduction of linear programming solvers in Oz is based on real-interval constraints introduced in Section 2.2.

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

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

or by

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

as part of a functor definition.

The module LP uses per default *LP_SOLVE 2.x* as linear programming solver. A version compatible with Mozart Oz can be downloaded via:

- http://www.mozart-oz.org/download/mozart-ftp/extras/lp_solve_2.3_mozart.tar.gz[2].

Unpack the archive and make it. You will be told what else has to be done. Please note that we are not able to sort out any problems concerning the actual *LP_SOLVE 2.x* solver and that we are not responsible for that kind of problems resp. bugs.

---

[1] The modules LP and RI are *not* provided on any Windows platform.
[2] http://www.mozart-oz.org/download/mozart-ftp/extras/lp_solve_2.3_mozart.tar.gz

Linear programming solver (LP solver) handle problems of the following kind [3]:

minimize resp. maximize:

$$c_1 x_1 \quad + \quad \ldots \quad + \quad c_n x_n \qquad \text{objective function}$$

subject to:

$$a_{1,1}x_1 \quad + \quad \cdots \quad + \quad a_{n,1}x_n \quad \diamond \quad b_1 \qquad \text{constraints}$$
$$\vdots \qquad\qquad \vdots \qquad \vdots$$
$$a_{1,m}x_1 \quad + \quad \cdots \quad + \quad a_{n,m}x_n \quad \diamond \quad b_m$$
$$(\diamond \in \{\leq, =, \geq\})$$

$$l_i \leq x_i \leq u_i \quad (i = 1, \ldots, n) \qquad \text{bound constraints.}$$

The module `LP` provides a procedure `LP.solve` to call an LP solver. Further, a procedure to configure the LP solver is provided (see Section *The Module* `LP`, *(The Mozart Constraint Extensions Reference)*).

A simple example.   A simple example explains best how the LP solver is invoked:

```
declare
X1 = {RI.var.bounds 0.0 RI.sup}
X2 = {RI.var.bounds 0.0 RI.sup}
Ret Sol
in
{LP.solve
 [X1 X2]
 objfn(row: [8.0 5.0] opt: max)
 constrs(
    constr(row: [1.0 1.0] type: '=<' rhs:6.0)
    constr(row: [9.0 5.0] type: '=<' rhs:45.0))
 Sol
 Ret}
```

The corresponding linear program is as follows:

$$
\begin{array}{rrcl}
\textbf{maximize:} & 8x_1 + 5x_2 & & \\
\textbf{subject to:} & x_1 + x_2 & \leq & 6 \\
& 9x_1 + 5x_2 & \leq & 45 \\
& x_1, x_2 & \geq & 0
\end{array}
$$

Note that the bound constraints for the LP solver are derived from the current bounds of the real-interval variables. Further, when minimizing the objective function the following constraint $c_1 x_1 + \ldots + c_n x_n \leq \overline{\texttt{Sol}}$ is added. On the other hand, the constraint $c_1 x_1 + \ldots + c_n x_n \geq \underline{\texttt{Sol}}$ is added when maximizing.

Before running the LP solver, the variables are constrained to

```
X1<real interval:[0, 1.79769e+308]>
```

and

```
X2<real interval:[0,1.79769e+308]>.
```

The LP solver binds the variables to: `X1=3.75`, `X2=2.25`, `Sol=41.25`, and `Ret=optimal`.

The tutorial problem: Solving a multiknapsack problem.   This tutorial uses a multiknapsack problem to demonstrate the benefits of combining finite domain constraint programming and linear programming. First, we tackle the problem with finite domain constraints and linear programming separately (see Section 3.2 and Section 3.3). One difficulty arises for linear programming: since integral solutions are required and the LP solver returns non-integral solution, we have to implement a branch&bound solver to obtain an integral solution. Finally, we combine both solvers.

Throughout this tutorial, we use a multi-knapsack problem (taken from [2]. The problem variables **x** represent the number of goods to be produced. Each good requires certain resources: man power, materials, and machines (represented by matrix **A**) where a given capacity per resource (**b**) may not be exceeded. Each good generates a profit according to **c** where the overall profit shall be maximal.

$$
\begin{aligned}
\textbf{maximize:} \quad & \mathbf{c} \ \times \ \mathbf{x} \\
\textbf{subject to:} \quad & \mathbf{A} \ \times \ \mathbf{x} \ \leq \ \mathbf{b} \\
& \mathbf{x} \text{ are integral.}
\end{aligned}
$$

where

$$
\mathbf{A} = \begin{bmatrix}
\textit{man:} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\textit{material 1:} & 0 & 4 & 5 & 0 & 0 & 0 & 0 & 4 & 5 & 0 & 0 & 0 & 0 \\
\textit{material 2:} & 4 & 0 & 3 & 0 & 0 & 0 & 3 & 0 & 4 & 0 & 0 & 0 & 0 \\
\textit{machine 1:} & 7 & 0 & 0 & 6 & 0 & 0 & 7 & 0 & 0 & 6 & 0 & 0 & 0 \\
\textit{machine 2:} & 0 & 0 & 0 & 4 & 5 & 0 & 0 & 0 & 0 & 5 & 4 & 0 & 0 \\
\textit{machine 3:} & 0 & 0 & 0 & 0 & 4 & 3 & 0 & 0 & 0 & 0 & 4 & 2 & 1 \\
\textit{machine 4:} & 0 & 3 & 0 & 0 & 0 & 5 & 0 & 3 & 0 & 0 & 0 & 3 & 3
\end{bmatrix}
\mathbf{b} = \begin{bmatrix}
14 \\ 17 \\ 20 \\ 34 \\ 26 \\ 16 \\ 16
\end{bmatrix}
$$

$$
\mathbf{c} = \begin{bmatrix} 5 & 7 & 5 & 11 & 8 & 10 & 6 & 8 & 3 & 12 & 9 & 8 & 4 \end{bmatrix}
$$

$$
\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} & x_{12} & x_{13} \end{bmatrix}
$$

The problem specification as Oz term is as follows:

```
declare
Problem =
problem(
   resources:
      resource(
         man:       r(ta: 14 npp: [1 1 1 1 1 1 1 1 1 1 1 1 1])
         material1: r(ta: 17 npp: [0 4 5 0 0 0 0 4 5 0 0 0 0])
         material2: r(ta: 20 npp: [4 0 3 0 0 0 3 0 4 0 0 0 0])
         machine1:  r(ta: 34 npp: [7 0 0 6 0 0 7 0 0 6 0 0 0])
         machine2:  r(ta: 26 npp: [0 0 0 4 5 0 0 0 0 5 4 0 0])
         machine3:  r(ta: 16 npp: [0 0 0 0 4 3 0 0 0 0 4 2 1])
```

```
            machine4:  r(ta: 16 npp: [0 3 0 0 0 5 0 3 0 0 0 3 3]))
    profit: [5 7 5 11 8 10 6 8 3 12 9 8 4])
```

3.2   The Finite Domain Model

The finite domain model is a one-to-one translation of the LP model. Every problem
variable of **x** is represented by a finite domain variable. The inequalities are expressed
by appropriate finite domain constraints.

```
declare
fun {KnapsackFD Problem}
   NumProducts = {Length Problem.profit}
   Resources   = Problem.resources
in
   proc {$ Sol}
      sol(maxprofit: MaxProfit = {FD.decl}
          products: Products = {FD.list NumProducts 0#FD.sup})
      = Sol
   in
      MaxProfit = {FD.sumC Problem.profit Products '=:'}

      {ForAll {Arity Resources}
       proc {$ ResourceName}
          Resource = Resources.ResourceName
       in
          {FD.sumC Resource.npp Products '=<:' Resource.ta}
       end}

      {FD.distribute naive Products}
   end
end
```

The function `KnapsackFD` returns a procedure abstracting the script. The solution vari-
able `Sol` of the script is constrained to a record. The record provides access to the
individual quantities of the individual products (under feature `products`) to obtain a
maximum profit (under feature `maxprofit`).

The variable `Products` refers to a list of finite domain problem variables (correspond-
ing to $[x_1,\ldots,x_n]^T$ in the LP model) and `MaxProfit` is constrained to be the scalar
product of the `Product` variable and the profit vector for the problem specification
(see Section 3.1).

The `ForAll` iterator imposes the inequality constraints $a_{1,i}x_1 + \cdots + a_{n,i}x_n \le b_i$ to the
problem variables. The distribution strategy is straightforwardly chosen to `naive`. Ex-
perimenting with first fail (`ff`) produced even worse results.

Solving the problem by calling

```
{ExploreBest {KnapsackFD Problem}
             proc {$ O N} O.maxprofit <: N.maxprofit end}
```

produces the following search tree.
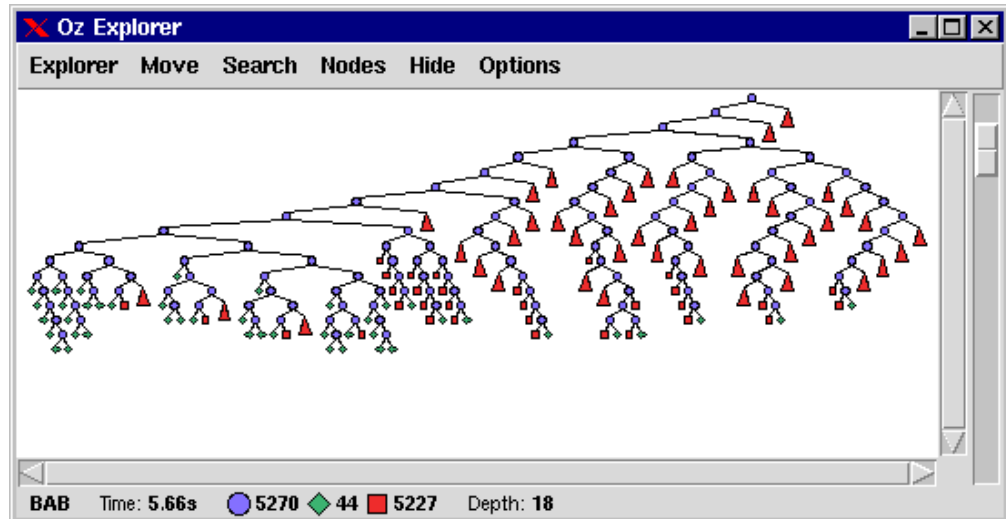
3.3 The Linear Programming Model

Tackling a multi-knapsack problem with a LP solver amounts to implementing a branch & bound solver to obtain integral solutions. The idea is to compute a continuous solution and to branch over the problem variables with continuous solutions. This is done until only integral problem variables are left. This is what the procedure DistributeKnapSackLP does.

```
declare
proc {DistributeKnapSackLP Vs ObjFn Constraints MaxProfit}
   choice
      DupVs = {DuplicateRIs Vs}
      DupMaxProfit V DupV
   in
      DupMaxProfit = {RI.var.bounds
                        {RI.getLowerBound MaxProfit}
                        {RI.getUpperBound MaxProfit}}

      {LP.solve DupVs ObjFn Constraints DupMaxProfit optimal}

      V#DupV = {SelectVar Vs#DupVs}

      case {IsDet V} then
         DupMaxProfit = MaxProfit
         DupVs        = Vs
      else
         choice
            {RI.lessEq {Ceil DupV} V}
```

```
            []
               {RI.lessEq V {Floor DupV}}
            end
            {DistributeKnapSackLP Vs ObjFn Constraints MaxProfit}
         end
      end
end
```

It first duplicates the problem variables (note this is possible due to stability) and invokes the LP solver on them to compute a (possibly continuous) solution. Then it selects the first duplicated continuous problem variable `DupV` by `SelectVar` (see below). If continuous variables are left (see the **else** branch of the **case** statement), it creates two choices on the corresponding original problem variable v: $\lceil DupV \rceil \leq V \vee V \leq \lfloor DupV \rfloor$ and calls `DistributeKnapSackLP` recursively. In case no continuous variables are left, an integral solution is found and the original problem variables are unified with duplicated ones.

For completeness sake the auxiliary functions `SelectVar` and `DuplicateRIs` are presented here.

```
declare                              declare
fun {SelectVar VsPair}               fun {DuplicateRIs Vs}
   case VsPair                          {Map Vs
   of nil#nil then unit#unit            fun {$ V}
   [] (VH|VT)#(RVH|RVT) then               {RI.var.bounds
      % check for integrality              {RI.getLowerBound V}
      case RVH == {Round RVH}              {RI.getUpperBound V}}
      then {SelectVar VT#RVT}           end}
      else VH#RVH end                 end
   else unit
   end
end
```

The procedure `KnapsackLP` return the script which creates the appropriate parameters for the LP solver and eventually calls `DistributeKnapSackLP`.

```
declare
fun {KnapsackLP Problem}
   NumProducts = {Length Problem.profit}
   Resources   = Problem.resources
in
   proc {$ Sol}
      sol(maxprofit: MaxProfit = {RI.var.decl}
          products: Products = {MakeList NumProducts})
      = Sol

      ObjFn Constraints
   in
```

```
{ForAll Products proc {$ V}
                       {RI.var.bounds 0.0 RI.sup V}
                    end}


ObjFn = objfn(row: {Map Problem.profit IntToFloat}
              opt: max)


Constraints =
{Map {Arity Resources}
 fun {$ ResourceName}
    Resource = Resources.ResourceName
 in
    constr(row: {Map Resource.npp IntToFloat}
           type: '=<'
           rhs: {IntToFloat Resource.ta})
 end}


{DistributeKnapSackLP Products ObjFn Constraints
 MaxProfit}
   end
 end
```
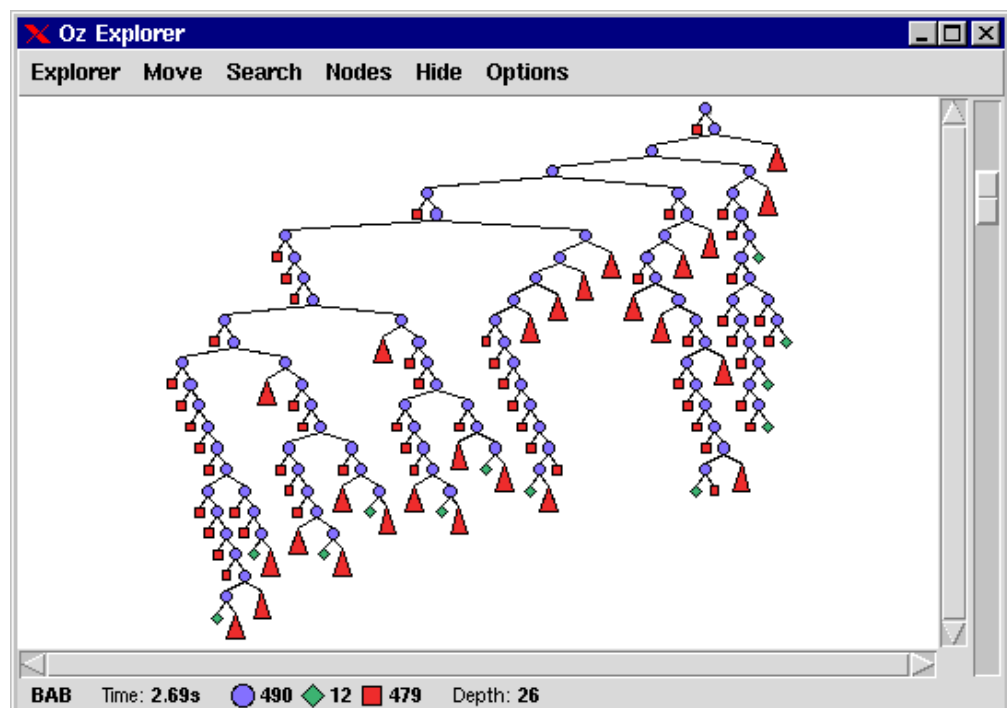
Feeding

```
{ExploreBest {KnapsackLP Problem}
             proc {$ O N}
                {RI.lessEq O.maxprofit+1.0 N.maxprofit}
             end}
```

produces the following search tree.

3.4 Combining Both Models

Combining both models is simply done by adding the finite domain model without distribution to the linear programming model. The propagator `{RI.intBounds F I}` is used to connect real-interval constraints with finite domain constraints. It constrains `F` and `I` to denote the same integer either as float or as integer, respectively.

```
declare
fun {KnapsackFDLP Problem}
   NumProducts = {Length Problem.profit}
   Resources   = Problem.resources
in
   proc {$ Sol}
      sol(maxprofit: FDMaxProfit = {FD.decl}
          products: FDProducts = {FD.list NumProducts 0#FD.sup})
      = Sol

      ObjFn Constraints
      MaxProfit = {RI.var.decl}
      Products  = {MakeList NumProducts}
   in
      %
      % finite domain constraints part
      %
      FDMaxProfit = {FD.sumC Problem.profit FDProducts '=:'}

      {ForAll {Arity Resources}
       proc {$ ResourceName}
          Resource = Resources.ResourceName
       in
          {FD.sumC Resource.npp FDProducts '=<:' Resource.ta}
       end}

      %%
      %% linear programming part
      %%
      {ForAll Products
       proc {$ V} {RI.var.bounds 0.0 RI.sup V} end}

      ObjFn = objfn(row: {Map Problem.profit {IntToFloat I}}
                    opt: max)

      Constraints =
      {Map {Arity Resources}
        fun {$ ResourceName}
          Resource = Resources.ResourceName
       in
          constr(row: {Map Resource.npp IntToFloat}
                 type: '=<'
```

```
                                        rhs: {IntToFloat Resource.ta})
                  end}


                  %%
                  %% connecting both constraint systems
                  %%
                  {RI.intBounds MaxProfit FDMaxProfit}
                  {Map Products
                   proc {$ R D} {RI.intBounds R D} end FDProducts}



                  {DistributeKnapSackLP Products ObjFn Constraints
                   MaxProfit}
            end
      end
```
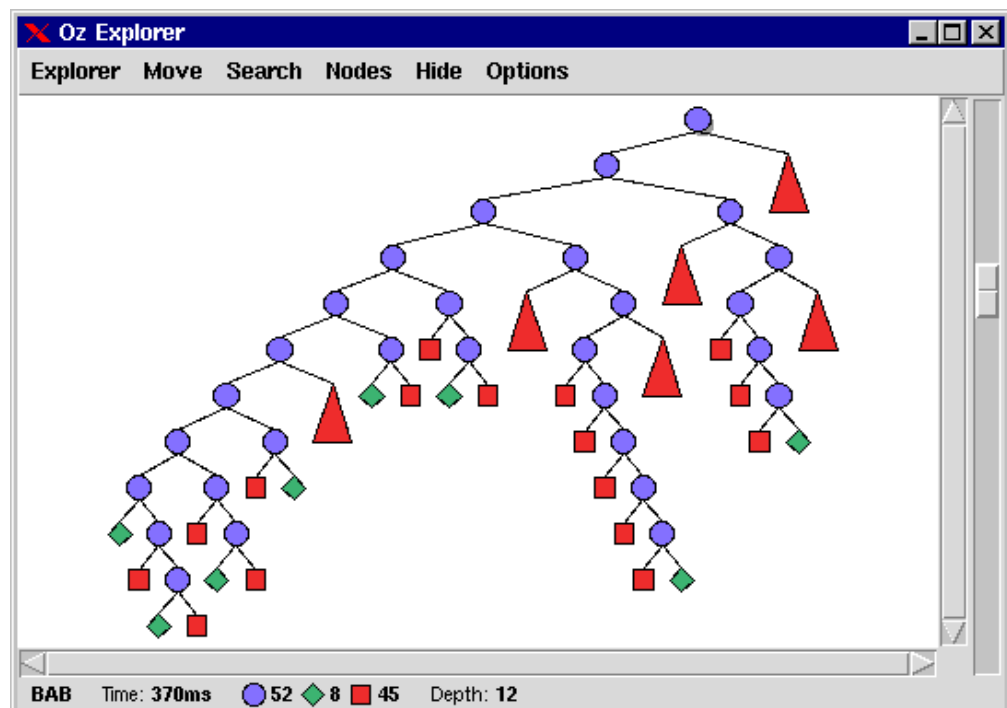
The Oz Explorer produces the following search tree by calling

```
{ExploreBest {KnapsackFDLP Problem}
              proc {$ O N} O.maxprofit <: N.maxprofit end}
```

.

3.5   Short Evalution

The following table shows impressively the benefits of combining propagation-based and linear programming solvers for this kind of problem. We used *LP_SOLVE 2.0* as LP solver. Note using a different LP solver may produce different results. By

combining both constraint models the number of nodes in the search tree could be reduced by two resp. one to orders of magnitudes. This results in a speed-up of one order of magnitude and memory saving by the same amount.

Model	Nodes	Sols	Failures	Depth	runtime [sec]	heap [MB]
FD Model	5270	44	5227	18	4.980	10.4
LP Model	490	12	479	26	3.390	3.3
FD+LP Model	52	8	45	12	0.390	0.6

The times were taken on a Pentium Pro 200MHz with 192 MB memory.

The described technique has been used to tackle set partitioning problems. In contrast to [8] all problem could be solve in acceptable time (detailed benchmarks are included as they are available).

Bibliography

[1] Frédéric Benhamou. Interval constraint logic programming. In Andreas Podelski, editor, *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, pages 1–21. Springer Verlag, 1995.

[2] H. Beringer and B. de Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plümer, editors, *Logic programming: Formal methods and practical applications*, pages 245–272. Elsevier, 1995.

[3] Vašek Chvátal. *Linear Programming*. W.H. Freeman and Company, 41 Madison Avenue, New York 10010, 1983.

[4] Hartmut Schwab. *Documentation for lp_solve*.

[5] Christian Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification*. PhD thesis, Technisch-Naturwissenschaftliche Fakultät der Technischen Universität Wien, October 1990.

[6] ILOG, Inc. CPLEX Devision. *Using the CPLEX Callable Library Version 5.0*, 1997.

[7] Christian Schulte Michael Mehl, Ralf Scheidhauer. *An abstract Machine for Oz*. Programming Systems Lab, Saarbrücken, March 1995.

[8] Tobias Müller. Solving set partitioning problems with constraint programming. In *Proceedings of the Sixth International Conference on the Practical Application of Prolog and the Forth International Conference on the Practical Application of Constraint Technology – PAPPACT98*, pages 313–332, London, UK, March 1998. The Practical Application Company Ltd.

[9] Robert B. Murray. *C++ Strategies and Tactics*. Programming Systems Lab, Saarbrücken, March 1993.

[10] Roland H. Pesch Richard M. Stallman. *Debugging with GDB: The GNU Source-Level Debugger version 4.7*. Programming Systems Lab, Saarbrücken, October 1992.

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

Index