Finite Set Constraints in Oz

Tobias Müller and Martin Müller
Forschungsbereich Programmiersysteme
Universität des Saarlandes
66041 Saarbrücken, Germany
{tmueller,mmueller}@ps.uni-sb.de

Abstract

We report on the extension of the concurrent constraint language Oz by constraints over finite sets of integers. Set constraints are an important addition to the constraint programming system Oz and are very employable in natural language processing and general problem solving. This extension profits much from its integration with the existing constraint systems over finite domains and feature trees, as well as from the availability of first-class procedures. This combination of features is unique to Oz. This paper focuses on the expressiveness gained by set constraints and on the benefits of the integration with finite domain constraints. A number of case studies demonstrates programming techniques exploring these advantages.

1 Introduction

Constraints over finite sets of integers (for short set constraints) are very employable in combinatorial problem solving and in natural language processing. Sets are a natural and frequent data structure in many problems, *e.g.* set constraints allow to conveniently express certain type hierarchies as used in unification grammars.

The higher-order concurrent constraint language Oz [27, 21] already supports two powerful constraint systems; one over finite domain constraints [17] as a well-established constraint system for combinatorial problem solving, and one over feature trees (*viz.* records [23]) as required by applications in natural language processing.

Supplementing both constraint systems with set constraints yields a significant gain in expressiveness for the mentioned classes of applications. The integration of set constraints into the general-purpose programming system Oz further empowers Oz as a programming platform for a wide range of constraint problems. The additional expressiveness of this host language provided by first-class procedures, inference engines [25], and objects [10] is very convenient.

The implementation effort for the integration of set constraints into Oz was less than three man-months using the C++ constraint propagator interface of Oz [18] and resulted in a library with adequate efficiency.

This paper demonstrates the gained expressiveness by example. We show a number of programming techniques and point out conciseness of problem formulations. We also stress that the combination of set constraints and finite domain constraints improves the problem solving capabilities by extra constraint propagation and new patterns to avoid symmetric solutions in problem formulations.

To support this claim, we investigate a new constraint which associates the n minimal elements of a set with n finite domain variables. We employ this constraint in a set-based implementation of the Steiner problem [3]. We show that in contrast to the straightforward formulation a significant reduction of choice points is obtained (due to improved constraint propagation) which results in a notable decrease of heap space consumption and runtime.

Plan of the paper. The paper is structured as follows. Section 2 introduces terminology and the choice of set constraints we support in Oz. The following Sections 3 and 4 discuss programming techniques employing set constraints; Section 4 in particular focuses on the interaction of set constraints with finite domain constraints and Section 5 discusses Oz-specific programming idioms. Section 6 surveys related work, and Section 7 briefly summarizes.

2 Finite Set Intervals in Oz

We assume an integer constant *sup* and consider subsets of the finite universal set $\mathcal{U} = \{0,\ldots,sup\}$.[1] We adopt the convention that *constants* and *variables* are denoted by lower *resp.* upper case letters. A lower case n denotes an *integer constant* being element of $\mathcal{U}$; s designates a *set constant* ranging over subsets of $\mathcal{U}$; N denotes a *finite domain integer variable* to be interpreted as an element in $\mathcal{U}$, and S is a *set variable* interpreted as a subset of $\mathcal{U}$.

Amongst the constraints we distinguish between basic and non-basic ones. *Basic constraints* are those constraints for which satisfiability and entailment (*i.e.*, logic implication) can be decided efficiently (or at least efficient enough for the intended applications). *Non-basic constraints* are those for which we intend *resp.* have to treat satisfiability and entailment in an incomplete fashion because of their computational complexity.

Basic and Non-basic Constraints. The most important basic constraints $\mathcal{B}$ provide lower and upper bounds for a set variable:[2]

$$\mathcal{B} \quad ::= \quad \mathcal{B}_1 \wedge \mathcal{B}_2 \quad | \quad s \subseteq S \quad | \quad S \subseteq s \quad | \quad \ldots$$

The constraints $n \in S$ and $n \notin S$ are derived forms since $n \in S \leftrightarrow \{n\} \subseteq S$ and $n \notin S \leftrightarrow S \subseteq \mathcal{U}\backslash\{n\}$ (where $\mathcal{U}\backslash\{n\}$ is finite since $\mathcal{U}$ is finite). The cardinality of a set constant s is denoted by $\#s$. A second pair of basic constraints provides bounds on the *cardinality* $\#S$ of (the denotation of) a set variable S:

$$\mathcal{B} \quad ::= \quad \ldots | \quad n \leq \#S \quad | \quad \#S \leq n$$

All other set operations like union $\cup$, intersection $\cap$, asymmetric difference $\backslash$, disjointness $\|$, etc., are non-basic constraints $\mathcal{C}$:

$$\begin{array}{lll} \mathcal{C} & ::= & \mathcal{B} \quad | \quad \mathcal{C}_1 \wedge \mathcal{C}_2 \quad | \quad S = \mathcal{E} \quad | \quad S_1 \subseteq S_2 \quad | \quad S_1 \| S_2 \quad | \quad \ldots \\ \mathcal{E} & ::= & S_1 \cup S_2 \quad | \quad S_1 \cap S_2 \quad | \quad S_1 \backslash S_2 \quad | \quad \ldots \end{array}$$

Solved Forms. Every basic constraint $\mathcal{B}$ can easily be checked for satisfiability; further, if $\mathcal{B}$ is satisfiable, it can be brought into a solved form which contains for every set variable S the greatest (least) sets s_{glb}[3] (s_{lub}[4]) and integers n_{min} and n_{max} such that $\mathcal{B}$ entails

$$s_{glb} \subseteq S \subseteq s_{lub} \quad \wedge \quad n_{min} \leq \#S \leq n_{max} \,.$$

[1] The set notation $\{a,\ldots,b\}$ always denotes a *convex* set. *I.e.*., $c \in \{a,\ldots,b\}$ whenever $a \leq c \leq b$.

[2] In the notation of [9], $s \subseteq S$ and $S \subseteq s$ appear as $s \in_{[\emptyset,S]}$ and $s \in_{[S,\mathcal{U}]}$, respectively.

[3] **g**reatest **l**ower **b**ound

[4] **l**east **u**pper **b**ound

Obviously, $\#s_{glb} \leq n_{min}$ and $n_{max} \leq \#s_{lub}$ are properties of the solved form. Given a basic constraint $\mathcal{B}$ in solved form, we denote with $glb(S)$ and $lub(S)$ (*w.r.t.* $\mathcal{B}$) the respective bounds of the set interval for a variable S. A satisfiable constraint *determines* a (set) variable S if the denotation of S is uniquely fixed by the constraint. A basic constraint $\mathcal{B}$ in solved form determines a variable S if and only if $glb(S) = lub(S)$ *w.r.t.* $\mathcal{B}$.

Constraint Store and Propagators. The concurrent constraint framework [15, 24, 27] organizes computation in a number of concurrent actors operating and communicating over a shared constraint store. The implementation model underlying constraint programming in Oz [21] places basic constraints in the *constraint store* and implements non-basic constraints as actors, called *propagators*.

The operation to *tell* a basic constraint $\mathcal{B}$ to the constraint store $\mathcal{C}$ is executed as follows: If $\mathcal{C} \wedge \mathcal{B}$ is satisfiable, then the store $\mathcal{C}$ is extended to $\mathcal{C} \wedge \mathcal{B}$ without interruption; otherwise, a failure condition is raised. A *propagator* P is an actor whose logic semantics is given by a (usually, non-basic) constraint $\mathcal{C}_P$, and whose operational semantics is correct but not necessarily complete *w.r.t.* its logic semantics. The variables in C_P are called the *parameters* of P. *Imposing* a propagator P means installing an actor which continuously watches the constraint store *w.r.t.* its parameters. When the parameters of P become further constrained in the store, P may be activated and then, on its part, tells constraints according to its operational semantics. Once the constraint store entails $\mathcal{C}_P$, P is redundant and may disappear. P will disappear latest when the constraint store determines all its parameters, or if $\mathcal{C}_P$ is inconsistent with the constraint store (in which case a failure condition is raised). *Propagation* is the process of running all propagators to termination.

As an example for constraint propagation, assume the store $\emptyset \subseteq S_1, S_2 \subseteq \{1,\ldots,5\}$, as well as propagators for the non-basic constraints $S_1 \cup S_2 = \{1,\ldots,5\}$ and $S_1 \| S_2$. Telling the basic constraints $1 \in S_1$ and $2 \notin S_2$ yields the intermediate constraint store $\{1\} \subseteq S_1 \subseteq \{1,\ldots,5\}$ and $\emptyset \subseteq S_2 \subseteq \{1,3,4,5\}$. Then, when constraint propagation has finished, the constraint store holds $\{1,2\} \subseteq S_1 \subseteq \{1,\ldots,5\} \wedge \emptyset \subseteq S_2 \subseteq \{3,4,5\}$. Eventually, telling $\#S_2 = 3$ determines S_1 and S_2, *i.e.*, yields the constraint store $S_1 = \{1,2\} \wedge S_2 = \{3,4,5\}$.

Distribution. Problem solving in Oz is realized as an interleaving of (basic) constraint solving and propagation, as well as search guided by the creation of choice points and distribution. A typical choice point for set constraint problems is given by a disjunction of the form $n \in S \vee n \notin S$. *Distribution* is the operation of picking one of the current choice points and offering the suggested alternatives for independent exploration [26]. Which choice point to pick at a certain stage and in which order to explore the alternatives is up to the *distribution strategy* fixed for the problem at hand. As part of a search procedure, the above-mentioned failure conditions are interpreted as the absence of solutions; elsewhere they amount to a runtime error.

Connecting Finite Domains and Finite Sets. The cardinality constraint $N = \#S$ relates sets and integers. The presence of the finite domain constraint system of Oz [17, 21] suggests to generalize the basic constraints $n \leq \#S$ and $\#S \leq n$ to the more expressive propagators

$$N \leq \#S \quad | \quad \#S \leq N.$$

This establishes a close link between both constraint systems since the integer variables can, *e.g.* carry domain restrictions like $N \in \{1,\ldots,5\}$. Often it is also convenient to realize that we have finite sets of *integers* (instead of arbitrary elements) and to exploit their natural order or the expressiveness of arithmetics. As an example of propagators for constraints required by a natural language parsing application, consider the propagators $min(S) = N$ and $max(S) = N$ meaning that the minimal (*resp.* maximal) integer in (the denotation of) S equals (the denotation of) N. A propagator which turned out useful to break symmetries and to provide extra propagation (see Section 4.1) generalizes $min(S) = N$ to $minN(S) = [N_1, \ldots, N_k]$ saying that $k \leq \#S$ and that the k minimal elements of S are N_1 through N_k in this order.

3 Problem Solving with Set Constraints

This section illustrates problem solving with finite set constraints in Oz by means of examples. The provided constraint solving abstractions of Oz require that problems are to be formulated as unary Oz procedures, typically with the following structure.

```
proc {Problem Sol}
   ...  % impose constraints and propagators
   ...  % specify distribution strategy
end
```

These are then passed to an inference engine (*e.g.* the Oz Explorer [25]) to explore the specified search space. The formal argument `Sol` refers to the sought solution.

Note some examples in this section use finite domain constraints. Finite domain infix operators end with a "`:`", as for instance "`=<:`". Finite domain library procedures begin with "`FD.`", as for instance "`FD.decl`".

3.1 The Steiner Problem

The ternary Steiner problem [3] of order n asks for $n(n-1)/6$ sets with cardinality 3 such that every two of them share at most one element. It has been proved that for a solution to exist n *mod* 6 must be 1 or 3. The Oz function `Steiner` below maps an integer n to an (anonymous) procedure modeling the Steiner problem of order n. (Anonymous procedures in Oz are marked with `$`.) To solve the Steiner problem of order 9, say, one may invoke the Oz Explorer by executing `{ExploreOne {Steiner 9}}`.

```
fun {Steiner N}
   proc {$ Ss}
      case
         N mod 6 == 1 orelse N mod 6 == 3             % 1.
      then
         {FS.var.list.lub (N*(N-1)) div 6 [1#N] Ss}  % 2.
         {ForAll Ss proc {$ S} {FS.card S 3} end}    % 3. #S = 3

         {ForAllTail Ss                              % 4.
          proc {$ S1|Sr}
             {ForAll Sr                              % 5.
              proc {$ S2} S3 in
                 S3 = {FS.intersect S1 S2}           % 6. S3 = S1 ∩ S2
                 {FS.cardRange 0 1 S3}               % 7. #S3 ∈ {0,1}
              end}
          end}

         {FS.distribute naive Ss}                    % 8.
      else fail end
   end
end
```

The case statement in line 1 checks whether there can be any solutions for a given N. The library procedure `{FS.var.list.lub Len Lub Xs}` constrains `Xs` to a list of `Len` set variables S_i (see line 2). Further, each of these is constrained by $\emptyset \subseteq S_i \subseteq set(\texttt{Lub})$ where $set(\texttt{Lub})$ is the set described by the Oz term `Lub`: *e.g.* $set(\texttt{[1 3 5\#7]}) = \{1,3,5,6,7\}$. The cardinality of all S_i is constrained to 3 (line 3). Two nested loops (lines 4 and 5), using the library procedures `ForAllTail` [5] and `ForAll` [6], require all pairwise distinct elements of `Xs` to have at most one element in common. The library abstraction `{FS.cardRange M N S}` imposes the constraint $\texttt{M} \leq \#\texttt{S} \leq \texttt{N}$, and `{FS.intersect X Y Z}` expectedly $\texttt{X} \cap \texttt{Y} = \texttt{Z}$ (lines 6 and 7). Line 8 specifies the distribution strategy to always pick the leftmost undetermined element `S` of `Ss` and the smallest integer n with $n \in lub(S) \backslash glb(S)$, and then to distribute the choice point $n \in S \vee n \notin S$.

3.2 Hamming Distance

This example uses sets to model bit strings. The problem is as follows: given integers n, b, and d, find n tuples $w \in \{0,1\}^b$ such that the hamming distance of all pairwise distinct tuples is at least d. For example:

```
declare B=5 D=2 N=16
```

The *hamming distance* $h(v,w)$ between two tuples $v = (v_1,\ldots,v_b)$ and $w = (w_1,\ldots,w_b)$ where $v,w \in \{0,1\}^b$ is defined as the number of positions i where $v_i \neq w_i$. We can model tuples $w \in \{0,1\}^b$ as sets $s_w \subseteq \{1,\ldots,b\}$ where $i \in s_w$ if and only if $w_i = 1$. Then the hamming distance between v and w is just

$$h(v,w) \quad = \quad b - \#(s_v \cap s_w) - \#(\{1,\ldots,b\} \backslash (s_v \cup s_w)).$$

The condition $h(v,w) \leq d$ for all v,w now codes as

```
proc {MinDist Sv Sw}
   AllBits = {FS.value.new [1#B]}                        % AllBits = {1,...,B}
   Common1s = {FS.intersect Sv Sw}                       % Common1s = Sv ∩ Sw
   Common0s = {FS.complIn {FS.union Sv Sw} AllBits}      % AllBits\(Sv ∪ Sw)
in
   {FS.card Common1s} + {FS.card Common0s} =<: B-D
end
```

such that the hamming problem can be modelled as follows.

```
proc {Hamming Ss}
   {FS.var.list.lub N [1#B] Ss}

   {ForAllTail Ss
    proc {$ S1|Sr}
       {ForAll Sr proc {$ S2} {MinDist S1 S2} end}
    end}

   {FS.distribute naive Ss}
end
```

4 Interaction with Finite Domain Constraints

Arithmetics on the elements of finite sets can be a powerful means to prune the search space and to avoid symmetries. Therefore, we support mixed propagators operating over both the set and the

[5] `{ForAllTail [X`$_1$ `...` `X`$_n$`] P}` reduces to `{P [X`$_1$ `...` `X`$_n$`]} {P [X`$_2$ `...` `X`$_n$`]} ... {P [X`$_n$`]}`.

[6] `{ForAll [X`$_1$ `...` `X`$_n$`] P}` reduces to `{P X`$_1$`} {P X`$_2$`} ... {P X`$_n$`}`.

integer domain. This is conveniently achieved since Oz provides for a full-fledged finite domain solver with various flexible propagators for integers arithmetics (see [21]).

4.1 Ordering Sets

Problem formulations asking for a collection of sets run the risk of having numerous symmetric solutions. This can be avoided if an order on sets is available. Such an order can, *e.g.*, be given in terms of an integer rank $rank(s)$ associated with every set s.

An immediate way to define a rank is to interpret the characteristic function of every set as a bit string *resp.* as a binary number.

$$(b_0, b_1, \ldots, b_{sup})_2 \quad \text{where } b_i \in \{0,1\} \text{ and } b_i = 1 \text{ iff } i \in s$$

For large sup, however, this function is impractical since it takes huge values of order $O(2^{sup})$. Further, the obtained constraint propagation is not satisfactory.

In case the cardinality of all relevant sets is fixed, say for some s to k such that $s = \{n_1, \ldots, n_k\}$, we can do much better by ordering the integers n_1 through n_k and interpreting them as a number to the base $sup + 1$.[7]

$$(n_1, \ldots, n_k)_{sup+1} \tag{1}$$

If we have references N_1 through N_k to the elements n_1 through n_k we can state the fact that they must be ordered through the finite domain integer propagators

$$N_1 \texttt{<:} \ N_2 \texttt{<:} \ \ldots \ \texttt{<:} \ N_k\,. \tag{2}$$

This gives strong constraint propagation whenever the bounds of some N_i are narrowed. The library procedure `{FS.minN S DV}` supports the rank function (2) more immediately. Its logic semantics is

$$\exists o \geq n: \ S{=}\{X_1, \ldots, X_n\} \cup \{X_{n+1}, \ldots, X_o\} \ \wedge \ X_1 {<} \ldots {<} X_o \ \wedge \ DV{=}[X_1, \ldots, X_n] \ .$$

Informally, `FS.minN` constrains the elements of the list `DV` to the n minimal elements of `S` and vice versa. The propagator for the rank function for subsets of $\{1, \ldots, N\}$ with (uniformly) fixed cardinality 3 can now be implemented as follows (where *FD.sup* is the implementation dependent maximal integer available in the Oz *FD* system).

```
proc {Rank S N ?U}                              % ? annotates U as output
  Xs = {FD.list 3 1#N}  [X1 X2 X3] = Xs         % X1,X2,X3 ∈ {1,...,N}
  N1 = N+1  N1N1 = N1*N1 in
  U = {FD.decl}                                 % U ∈ {0,...,FD.sup}
  {FS.minN S Xs}                                % relate S and Xs
  U =: N1N1 * X1 + N1 * X2 + X3                 % compute rank
end
```

We examine the effect of this rank function by reconsidering the Steiner problem. Add the following code right before line 8 to the function `Steiner` in Section 3.1.

```
local
   S1|Sr = Ss                       % 1. split list Ss in head and tail
   Xs = {FD.list 3 1#N}             % 2. Xs = {X1,X2,X3}, X1,X2,X3 ∈ {1...,N}
   N1 = N+1  N1N1 = N1*N1
in
   {FS.minN S1 Xs}                  % 3. initial value for FoldL
   {FoldL Sr                        % 4.
    proc {$ [X1 X2 X3] S2 ?Ys}      % 5. ? annotates Ys as output
```

[7]Note that this does not require all sets to have the *same* fixed cardinality!

```
      [Y1 Y2 Y3] = Ys in
      Ys = {FD.list 3 1#N}              % 6. Y1,Y2,Y3 ∈ {1,...,N}
      {FS.minN S2 Ys}
      N1N1*X1 + N1*X2 + X3 <: N1N1*Y1 + N1*Y2 + Y3    % enforce order
   end
   Xs                                   % pass first set as initial value to FoldL
   _}                                   % ignore result of FoldL
end
```

The code steps through the list `Ss` of set variables which is required to have at least one element (line 1). The `FoldL` statement in line 4 imposes on all pairs of adjacent set variables an ordering constraint according to the anonymous procedure in line 5. This anonymous procedure is derived from the above defined procedure `Rank` but is tailored for the combination with `FoldL`.

These (logically) redundant ordering constraints significantly reduce memory consumption and runtime for this problem: The speed-up factor for the Steiner problem of order 9 is 6.3 and memory consumption reduces by a factor of 5.9. Further, the number of choice points and failures drops drastically (see the table below).

We also compare our set-based implementation against an analogous implementation with finite domain integer constraints where sets are modelled with lists of (reified) 0/1-variables (basically encoding sets by their characteristic functions; the program code can be found in the Appendix). As expected, the number of choice points and failures does not differ between both implementations. However, we observe a significant advantage in time and space for the set-based solution. We expect this observation to remain true for other set-specific problems.

The table below compares Oz's set *resp.* finite domain constraints for the steiner problem. The figures for choice points and failures were obtained by the Oz Explorer [25]. The lines *steiner(n)** and *steiner(n)* refer to the implementations which do *resp.* do not use the redundant ordering constraints. Since we compare two solutions for one problem we prefer to give ratios rather than absolute figures for runtime and memory consumption.

problem	**sets**		**finite domains**		**runtime**	**memory**
	choice points	failures	choice points	failures	$\frac{\text{fd}}{\text{sets}}$	$\frac{\text{fd}}{\text{sets}}$
steiner(7)	20	6	20	6	–	–
steiner(7)*	15	1	15	1	–	–
steiner(9)	4545	452	4545	452	2.8	3.6
steiner(9)*	565	54	565	54	2.5	2.7

Entries of '–' are due to unmeasurable runtime *resp.* memory consumption since the problem is too small.

4.2 Sets with Attributed Elements

Many practical problems require to associate set elements with attributes such as weights. Such weights may denote the cost or benefit contributed by some element of a set and can be employed to specify an optimal solution. Weights can also model resource consumption in problems involving limited resources, as, for instance, in the bin-packing which we discuss here.

The bin-packing problem is to partition a number of items i with individual weights w_i in a minimal number of bins with uniformly limited capacity c. Each bin b can be represented as a set s_b of items such that the capacity constraint $c \geq \sum_{i \in s_b} w_i$ is respected. Based on this idea, we model the bin-packing problem as follows.

```
fun {BinPacking Weights Capacity}
   proc {$ Ss}
```

```
      LB = {FoldL Weights Number.´+´ 0} div Capacity % 1. min. num. of bins
      UB = {Length Weights}                          % 2. max. num. of bins
      Items = {List.number 1 UB 1}                   % 3. Items = [1, ..., UB]
      NbBins = {FD.int LB#UB}
   in
      {FD.distribute naive [NbBins]}                 % 4. choose number of bins

      {FS.var.list.lub NbBins Items Ss}              % 5. bins S: ∅ ⊆ S ⊆ set(Items)}
      {FS.partition Ss {FS.value.new Items}}         % 6. pack each item exactly once
      {ForAll Ss                                     % 7. for all bins
       proc {$ S} BL in
          {FS.reified.areIn Items S BL}              % 8. reflect membership
          {FD.sumC Weights BL ´=<:´ Capacity}        % 9. respect capacity
       end}

      {FS.distribute naive Ss}                       % 10. place items
   end
end
```

In order to implement the capacity constraint, this implementation uses the (so-called reified) constraint $(i \in s \wedge r = 1) \vee (i \notin s \wedge r = 0)$ which reflects the validity of $i \in s$ into r (see line 8). The library procedure `{FS.reified.areIn Es S Bs}` realizes this constraint: For all elements `E` of the list `Es`, the membership of `E` in `S` is reflected in a 0/1-variable in the list `Bs` (at the corresponding position). In line 9, the inequation `Weights * BL =<: Capacity` on the scalar product of `Weights` and `BL` is computed using the propagator `{FD.sumC W BL ´=<:´ C}` from the *FD* library. The propagator `{FS.partition Ss U}` enforces the elements of the list `Ss` of sets to represent a partition of the set `U`.

Note that the distribution strategy is two-dimensional. First, in line 4, the distribution strategy fixes the number of bins for one branch of the search tree, where the numbers are tried in ascending order to minimize the number of bins. Thereafter, the items are placed according to the distribution strategy specified in line 10 which straightforwardly tries to put the next item in the leftmost bin (in `Ss`).

5 Oz-specific Programming Idioms

As part of a *concurrent* constraint language there is further expressiveness to our finite set constraint system than what was mentioned up to here. In particular, entailment of basic constraints by the constraint store is treated properly. The following example illustrates entailment of membership and ground inclusion. Assume the constraint store $\emptyset \subseteq S_1, S_2 \subseteq \mathcal{U}$, and execute the following statement:

```
thread if {FS.subset S1 S2} then {Show yes} else {Show no} end end   % S1 ⊆ S2?
thread if {FS.include 1 S2} then {Show yes} else {Show no} end end   % 1 ∈ S2?
```

This creates two threads which concurrently wait for the constraint store to either entail or disentail (*i.e.*, entail the negation of) their guards. Initially, the constraint store neither entails nor disentails any of these guards, hence both conditionals suspend. Now we execute the following lines.

```
{FS.include 1 S2}          % 1 ∈ S2?
{FS.include 2 S2}          % 2 ∈ S2?
{FS.value.new [1#2] S1}    % S1 = {1,2}
```

Execution of the first statement tells $1 \in S_2$ and wakes up the second conditional which then outputs `yes`. The second statement tells $2 \in S_2$; the first conditional remains suspended. Finally, telling $S_1 = \{1,2\}$ yields the constraint store $S_1 = \{1,2\} \wedge \{1,2\} \subseteq S_2 \subseteq \mathit{FS.sup}$ which entails $S_1 \subseteq S_2$ and wakes up the first conditional.

The combination of conditionals with first-class procedures provides a flexible tool-box for the development of user-defined propagators. For instance, consider the iterator which adds an element with value n^2 to S2 for every integer $n \in S1$ between 1 and 5.

```
{Loop.for 1 5 1
 proc {$ N}
    thread if {FS.include N S1} then {FS.include N*N S2} end end
 end}
```

This particular scheme is supported by the library somewhat more conveniently and efficiently. The procedure {FS.forAllIn S P} applies procedure P to every element in the set S as soon as it becomes known.[8]

```
{FS.forAllIn S1
 proc {$ N}
    case N*N=<FS.sup then {FS.include N*N S2} else skip end
 end}
```

Often, *e.g.* in program analysis [20], one asks for the least solution (or the smallest set solutions) of a collection of set constraints. The corresponding Oz idiom looks as follows.

```
proc {Problem S}
   . . .                                  % 1. impose basic constraints and propagators
   choice                                 % 2. Finally (i.e. on stability) ...
      S = {FS.value.new {FS.reflect.glb S}} % 3. ... equate S with glb(S)
   end
end
```

This idiom specifies that a certain collection of constraints be solved before, as final operation, an undetermined set variable S is equated to its greatest lower bound.[9] It allows, for instance, a concise formulation of the constraint-based safety analysis of Palsberg and Schwartzbach [20] in Oz.

6 Related Work

Meanwhile, there have been a number of proposals for the integration of sets into constraint/logic programming. The various set constraint systems differ in syntax, *i.e.*, in the set description language, and in the power of the constraint solving mechanism they provide. Let us briefly mention the general lines of the different approaches. For a thorough overview see [9, 28].

The simplest set constraint systems allow for the description of finite ground sets by enumeration of their elements $\{1,2,3\}$, $\{1, f(a), 2\}$, or $\{1,\{2\},\{\{3\}\}\}$. Our approach belongs into this class, along with Gervet's CONJUNTO [8], the set constraint library of *ECL^{i}PS^e* [6], and ILOG SOLVER [11, 22], a commercial library to enable constraint programming in C++. This line of research focuses on solving simple set constraints like membership and equality, and treats more complex constraints with consistency techniques from constraint programming.

More complex systems provide for a regular set description language. This includes the early work by Walinsky on CLP(Σ^*) [29] which deals with regular sets of *words*, as well as Foster's more recent CLP(SC) [7] (proposed by Kozen [12]) which deals with regular sets of *trees*. Regular sets of trees have been particularly prominent in static program analysis (see [1, 19] for overviews and

[8]On a first view, the reader may ignore the difference between **if** and **case** here. On a second view note that the guard of an **if** is a *statement* which succeeds or fails while the guard of an **case** is an *expression* which evaluates to a boolean value. The choice of **case** in the example at hand is simpler and more efficient.

[9]We remark that this non-monotonic reflection operation may exhibit an inconsistency not realized before. Recall that propagator P may implement their logic semantics C_P incompletely and thus not realize non-satisfiability of C_P *w.r.t.* the current constraint store.

references) and several specialised solvers have been developed. In this domain, constraint solving usually means testing satisfiability of a constraint, or emptiness of a set variable in all solutions (or a distinguished solution) of a constraint.

A third approach allows set descriptions of the form $\{X,Y\}$ (also called *set terms*) where X and Y are variables denoting elements, and provide an associative, commutative, and idempotent unification procedure. This is the approach of systems like CLPS [14], {log} [5], and others [2, 13, 28]. Yet different approaches allow set comprehensions like $\{x \mid p(x)\}$ with an intensional semantics [4], or consider non-standard set domains for interpretation of cyclic set descriptions like $X = \{X, \{X\}\}$ [16].

In comparison with its closest relatives CONJUNTO and ILOG SOLVER, our set constraint system differs in the following aspects. While CONJUNTO deals with finite ground sets over the Herbrand universe (including power sets), we only treat sets of integers; the same holds for ILOG SOLVER. In CONJUNTO, a set may, once and for all, be associated with weights per element. In contrast, we consider weights as attributes of individual elements which are not part of the set constraints, and allow to have several attributes per element (*e.g.* weight and benefit of an item in context of the knapsack problem). To our knowledge, ILOG SOLVER does not actively support attributed elements, although it ought to be expressive enough to model them. We allow for attributes to be used as parameter of distribution strategies, an option which the library of ILOG SOLVER as is does not provide. Also note that neither CONJUNTO nor ILOG SOLVER are *concurrent* constraint languages; hence both only provide a satisfiability test only while we need to handle entailment, too.

7 Conclusion

Set constraints are well-suited to extend the expressiveness of constraint programming platforms and in particular, in conjunction with other constraint systems, as *e.g.* finite domain constraints. But there is also a significant performance improvement which is only possible by the combination of both constraint systems (compare Section 4.1). Due to the availability of a high-level C++ constraint propagator interface, the implementation of the set constraint system took only three man-months.

Acknowledgments. The authors would like to thank Carmen Gervet and Denys Duchier for interesting and useful discussions. Further, we are grateful to Jörg Würtz and the anonymous referees for their comments on earlier versions of the paper. Additionally, Jörg Würtz provided the initial finite domain formulation of the Steiner program. The research reported here has been supported by the Esprit Working Group CCL II (EP 22457) and the SFB 378 at the Universität des Saarlandes.

References

[1] A. Aiken and N. Heintze. Constraint-Based Program Analysis, 1995. Invited Lecture at the 22^{nd} ACM Symposium on Principles of Programming Languages.

[2] C. Beeri, S. Nagvi, O. Shmueli, and S. Tsur. Set Constructors in a Logic Database Language. *The Journal of Logic Programming*, pages 181–232, 1991.

[3] N. Beldiceanu. An example of introduction of global constraints in chip: Application to block theory problems. Technical Report TR-LP-49, ECRC, Munich, Germany, May 1990.

[4] P. Bruscoli, A. Dovier, E. Pontelli, and G. Rossi. Compiling intensional sets in CLP. In *International Conference on Logic Programming*, pages 647–661. The MIT Press, Jan. 1994.

[5] A. Dovier and G. Rossi. Embedding Extensional Finite Sets in CLP. In *International Logic Programming Symposium*, 1993.

[6] ECRC. *ECL^{i}PS^e, User Manual Version 3.5.2*, December 1996.

[7] J. Foster. CLP(SC): Implementation and Efficiency Considerations. In *Proceedings Workshop on Set Constraints, held in Conjunction with CP'96*, Boston, Massachusetts, 1996.

[8] C. Gervet. *Set Intervals in Constraint-Logic Programming: Definition and Implementation of a Language*. PhD thesis, Université de France-Compté, Sept. 1995. European Thesis.

[9] C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(2), 1997.

[10] M. Henz, G. Smolka, and J. Würtz. Object-Oriented Concurrent Constraint Programming in Oz. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 27–48. The MIT Press, Cambridge, MA, 1995.

[11] ILOG, URL: `http://www.ilog.com`. ILOG SOLVER *3.2, User Manual*, 1996.

[12] D. Kozen. Set Constraints and Logic Programming. In *1st International Conference on Constraints in Computational Logics*, number 845 in Lecture Notes in Computer Science, 1994. also Information and Computation, to appear.

[13] G. Kuper. *Logic Programming with Sets*. Academic Press, New York, N.Y., 1990.

[14] B. Legeard and E. Legros. Short Overview of the CLPS System. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, Aug. 1991.

[15] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference (Melbourne)*, pages 858–876, 1987.

[16] S. Manandhar. An Attributive Logic of Set Descriptions and Set Operations. In *Annual Meeting of the Association of Computational Linguistics*, 1994.

[17] T. Müller and J. Würtz. A survey on finite domain programming in Oz. In *Notes on the DFKI-Workshop: Constraint-Based Problem Solving, Technical report D-96-02*, Kaiserslautern, Germany, 1996.

[18] T. Müller and J. Würtz. Extending a concurrent constraint language by propagators. In J. Maluszynski, editor, *Proceedings of the International Logic Programming Symposium*. The MIT Press, 1997. To appear.

[19] L. Pacholski and A. Podelski. Set Constraints: A Pearl in Research on Constraints. In G. Smolka, editor, *3rd International Conference on Principles and Practice of Constraint Programming*, 1997. Tutorial Abstract.

[20] J. Palsberg and M. I. Schwartzbach. Safety Analysis versus Type Inference. *Information and Computation*, 1995.

[21] Programming Systems Lab, Universität des Saarlandes. The Oz Programming System, 1997. `http://www.ps.uni-sb.de/www/oz/`.

[22] J. F. Puget. Finite Set Intervals. In *Proceedings Workshop on Set Constraints, held in Conjunction with CP'96*, Boston, Massachusetts, 1996.

[23] P. V. Roy, M. Mehl, and R. Scheidhauer. Integrating Efficient Records into Concurrent Constraint Programming. In *International Symposium on Programming Language Implementation and Logic Programming*, Aachen, Germany, Sept. 1996. Springer-Verlag, Berlin, Germany.

[24] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.

[25] C. Schulte. Oz Explorer: A visual constraint programming tool. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, Leuven, Belgium, 8-11 July 1997. The MIT Press.

[26] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In *Second Workshop on Principles and Practice of Constraint Programming*, Orcas Island, Washington, USA, May 1994. Springer-Verlag.

[27] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, Germany, 1995.

[28] F. Stolzenburg. Membership-constraints and complexity in logic programming with sets. In F. Baader and K. U. Schulz, editors, *Frontiers in Combining Systems*, pages 285–302. Kluwer Academic, Dordrecht, The Netherlands, 1996.

[29] C. Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proceedings of the International Conference on Logic Programming*, pages 181–190, 1989.

A Finite Domain Implementation of the Steiner Problem

```
fun {Steiner N}
   proc {$ Triples}
      N1 = N+1 N1N1 = N1*N1
   in
      case N mod 6 == 1 orelse N mod 6 == 3 then
         % create list of triples which model set of cardinality 3
         Triples = {MakeList N*(N-1) div 6}
         {ForAll Triples proc{$ T} T = {FD.list 3 1#N} end}

         % triple elements must be different
         {ForAll Triples FD.distinct}

         %  all pairs in two different triples must be different
         {ForAllTail Triples proc{$ [T11 T12 T13]|Tr}
                                {ForAll Tr
                                 proc{$ [T21 T22 T23]}
                                    {FD.sum [T11 =: T21
                                             T11 =: T22
                                             T11 =: T23
                                             T12 =: T21
                                             T12 =: T22
                                             T12 =: T23
                                             T13 =: T21
                                             T13 =: T22
                                             T13 =: T23]
                                     ´=<:´ 1}
                                 end}
                             end}

         % order triple elements
         {ForAll Triples proc{$ [T1 T2 T3]} T1<:T2 T2<:T3 end}

         % impose order on triples
         {ForAllTail Triples proc{$ [T11 T12 T13]|Tr}
                                case Tr of nil then skip
                                []  [T21 T22 T23]|_
                                then
                                   N1N1*T11 + N1*T12 + T13
                                   <:
                                   N1N1*T21 + N1*T22 + T23
                                end
                             end}

         % create choice points
         {FD.distribute naive {Flatten Triples}}
      else fail
      end
   end
end
```