A MiniZinc Tutorial

Kim Marriott and Peter J. Stuckey
with contributions from Leslie De Koninck and Horst Samulowitz

Contents

1 Introduction

MiniZinc is a language designed for specifying constrained optimization and decision problems over integers and real numbers. A MiniZinc model does not dictate how to solve the problem although the model can contain annotations which are used to guide the underlying solver.

MiniZinc is designed to interface easily to different backend solvers. It does this by transforming an input MiniZinc model and data file into a FlatZinc model. FlatZinc models consist of variable declaration and constraint definitions as well as a definition of the objective function if the problem is an optimization problem. The translation from MiniZinc to FlatZinc is specializable to individual backend solvers, so they can control what form constraints end up in. In particular, MiniZinc allows the specification of global constraints by decomposition.

2 Basic Modelling in MiniZinc

In this section we introduce the basic structure of a MiniZinc model using two simple examples.

2.1 Our First Example

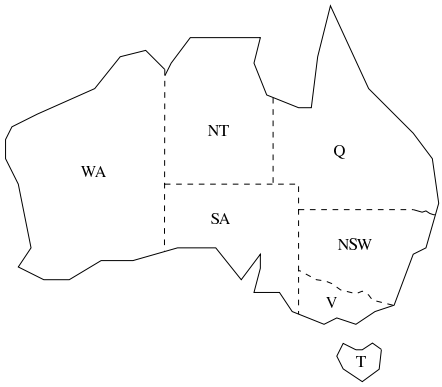


Figure 1: Australian states.

As our first example, imagine that we wish to colour a map of Australia as shown in Figure 1. It is made up of seven different states and territories each of which must be given a colour so that adjacent regions have different colours.

We can model this problem very easily in MiniZinc. The model is shown in Figure 2. The first line in the model is a comment. A comment starts with a '%' which indicates that the rest of the line is a comment. MiniZinc also has C-style block comments, which start with '/*' and end with '*/'.

The next part of the model declares the variables in the model. The line

```
int: nc = 3;
```

specifies a *parameter* in the problem which is the number of colours to be used. Parameters are similar to (constant) variables in most programming languages. They must be declared and given a type. In this case the type is **int**. They are given a value by an *assignment*. MiniZinc allows the assignment to be included as part of the declaration (as in the line above) or to be a separate assignment statement. Thus the following is equivalent to the single line above

```
int: nc;
nc = 3;
```

Unlike variables in many programming languages a parameter can only be given a *single* value, in that sense they are named constants. It is an error for a parameter to occur in more than one assignment.

AUST ☰ [DOWNLOAD]

```
% Colouring Australia using nc colours
int: nc = 3;

var 1..nc: wa;   var 1..nc: nt;  var 1..nc: sa;   var 1..nc: q;
var 1..nc: nsw;  var 1..nc: v;   var 1..nc: t;

constraint wa != nt;
constraint wa != sa;
constraint nt != sa;
constraint nt != q;
constraint sa != q;
constraint sa != nsw;
constraint sa != v;
constraint q != nsw;
constraint nsw != v;
solve satisfy;

output ["wa=\(wa)\t nt=\(nt)\t sa=\(sa)\n",
        "q=\(q)\t nsw=\(nsw)\t v=\(v)\n",
         "t=", show(t),  "\n"];
```

Figure 2: A MiniZinc model `aust.mzn` for colouring the states and territories in Australia.

The basic parameter types are integers (`int`), floating point numbers (`float`), Booleans (`bool`) and strings (`string`). Arrays and sets are also supported.

MiniZinc models can also contain another kind of variable called a *decision variable*. Decision variables are variables in the sense of mathematical or logical variables. Unlike parameters and variables in a standard programming language, the modeller does not need to give them a value. Rather the value of a decision variable is unknown and it is only when the MiniZinc model is executed that the solving system determines if the decision variable can be assigned a value that satisfies the constraints in the model and if so what this is.

In our example model we associate a *decision variable* with each region, wa, nt, sa, q, nsw, v and t, which stands for the (unknown) colour to be used to fill the region.

For each decision variable we need to give the set of possible values the variable can take. This is called the variable's *domain*. This can be given as part of the variable declaration and the type of the decision variable is inferred from the type of the values in the domain.

In MiniZinc decision variables can be Booleans, integers, floating point numbers, or sets. Also supported are arrays whose elements are decision variables. In our MiniZinc model we use integers to model the different colours. Thus each of our decision variables is declared to have the domain `1..nc` which is an integer range expression indicating the set $\{1, 2, ..., nc\}$

using the `var` declaration. The type of the values is integer so all of the variables in the model are integer decision variables.

Identifiers

Identifiers which are used to name parameters and variables are sequences of lower and uppercase alphabetic characters, digits and the underscore '_' character. They must start with a alphabetic character. Thus `myName_2` is a valid identifier. MiniZinc (and Zinc) *keywords* are not allowed to be used as identifier names, they are listed in Appendix A. Neither are MiniZinc *operators* allowed to be used as identifier names; they are listed in Appendix B.

MiniZinc carefully distinguishes between the two kinds of model variables: parameters and decision variables. The kinds of expressions that can be constructed using decision variables are more restricted than those that can be built from parameters. However, in any place that a decision variable can be used, so can a parameter of the same type.

Integer Variable Declarations

An integer parameter variable is declared as either:

`int` : ⟨*var-name*⟩
⟨*l*⟩ `..` ⟨*u*⟩ : ⟨*var-name*⟩

where l and u are fixed integer expressions.
An integer decision variable is declared as either:

`var int` : ⟨*var-name*⟩
`var` ⟨*l*⟩ `..` ⟨*u*⟩ : ⟨*var-name*⟩

where l and u are fixed integer expressions.

Formally the distinction between parameters and decision variables is called the *instantiation* of the variable. The combination of variable instantiation and type is called a *type-inst*. As you start to use MiniZinc you will undoubtedly see examples of *type-inst* errors.

The next component of the model are the *constraints*. These specify the Boolean expressions that the decision variables must satisfy to be a valid solution to the model. In this case we have a number of not equal constraints between the decision variables enforcing that if two states are adjacent then they must have different colours.

Relational Operators

MiniZinc provides the relational operators: equal (= or ==), not equal (!=), strictly less than (<), strictly greater than (>), less than or equal to (<=), and greater than or equal to (>=).

The next line in the model:

```
solve satisfy;
```

indicates the kind of problem it is. In this case it is a *satisfaction* problem: we wish to find a value for the decision variables that satisfies the constraints but we do not care which one.

The final part of the model is the *output* statement. This tells MiniZinc what to print when the model has been run and a solution is found.

Output and Strings

An output statement is followed by a *list* of strings. These are typically either string literals which are written between double quotes and use a C like notation for special characters, or an expression of the form *show(e)* where *e* is the MiniZinc expression. In the example `\n` represents the newline character and `\t` a tab.

There are also formatted varieties of *show* for numbers: *show_int(n,X)* outputs the value of integer X in at least $|n|$ characters, right justified if $n > 0$ and left justified otherwise; *show_float(n,d,X)* outputs the value of float X in at least $|n|$ characters, right justified if $n > 0$ and left justified otherwise, with d characters after the decimal point.

String literals must fit on a single line. Longer string literals can be split across multiple lines using the string concatenation operator ++ For example, the string literal `"Invalid datafile: Amount of flour is non-negative"` is equivalent to the string literal expression `"Invalid datafile: " ++ "Amount of flour is non-negative"`.

MiniZinc supports interpolated strings. Expressions can be imbedded directly in string literals, where a sub string of the form `\(e)` is replaced by the result of `show(e)`. For example `"t=\(t)\n"` produces the same string as `"t=" ++ show(t) ++ "\n"`.

A model can contain at most one output statement.

With the G12 implementation of MiniZinc we can evaluate our model by typing

```
$ mzn-g12fd aust.mzn
```

where `aust.mzn` is the name of the file containing our MiniZinc model. We must use the file extension ".mzn" to indicate a MiniZinc model. The command `mzn-g12fd` uses the G12 finite domain solver to evaluate our model.

When we run this we obtain the result:

```
wa=2  nt=3  sa=1
q=2  nsw=3  v=2
t=1
----------
```

The line of 10 dashes `----------` is output automatically added by the MiniZinc output to indicate a solution has been found.

2.2 An Arithmetic Optimisation Example

Our second example is motivated by the need to bake some cakes for a fete at our local school. We know how to make two sorts of cakes.[1] A banana cake which takes 250g of self-

[1] WARNING: please don't use these recipes at home

CAKES ☰ [DOWNLOAD]

```
% Baking cakes for the school fete

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% flour
constraint 250*b + 200*c <= 4000;
% bananas
constraint 2*b  <= 6;
% sugar
constraint 75*b + 150*c <= 2000;
% butter
constraint 100*b + 150*c <= 500;
% cocoa
constraint 75*c <= 500;

% maximize our profit
solve maximize 400*b + 450*c;

output ["no. of banana cakes = \(b)\n",
        "no. of chocolate cakes = \(c)\n"];
```

Figure 3: Model for determining how many banana and chocolate cakes to bake for the school fete.

raising flour, 2 mashed bananas, 75g sugar and 100g of butter, and a chocolate cake which takes 200g of self-raising flour, 75g of cocoa, 150g sugar and 150g of butter. We can sell a chocolate cake for $4.50 and a banana cake for $4.00. And we have 4kg self-raising flour, 6 bananas, 2kg of sugar, 500g of butter and 500g of cocoa. The question is how many of each sort of cake should we bake for the fete to maximise the profit. A possible MiniZinc model is shown in Figure 3.

The first new feature is the use of *arithmetic expressions*.

Integer Arithmetic Operators

MiniZinc provides the standard integer arithmetic operators. Addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`div`) and integer modulus (`mod`). It also provides `+` and `-` as unary operators.
Integer modulus is defined to give a result (a `mod` b) that has the same sign as the dividend a. Integer division is defined so that $a = b*(a$ `div` $b) + (a$ `mod` $b)$.
MiniZinc provides standard integer functions for absolute value (`abs`) and power function (`pow`). For example `abs(-4)` and `pow(2,5)` evaluate to 4 and 32 respectively.
The syntax for arithmetic literals is reasonably standard. Integer literals can be decimal, hexadecimal or octal. For instance `0`, `5`, `123`, `0x1b7`, `0o777`.

The second new feature shown in the example is optimisation. The line

```
solve maximize 400 * b + 450 * c;
```

specifies that we want to find a solution that maximises the expression in the solve statement called the *objective*. The objective can be any kind of arithmetic expression. One can replace the key word `maximize` by `minimize` to specify a minimisation problem.

When we run this we obtain the result:

```
no. of banana cakes = 2
no. of chocolate cakes = 2
----------
==========
```

The line `==========` is output automatically for optimisation problems when the system has proved that a solution is optimal.

2.3 Datafiles and Assertions

A drawback of this model is that if we wish to solve a similar problem the next time we need to bake cakes for the school (which is often) we need to modify the constraints in the model to reflect the ingredients that we have in the pantry. If we want to reuse the model then we would be better off to make the amount of each ingredient a parameter of the model and then set their values at the top of the model.

Even better would be to set the value of these parameters in a separate *data file*. MiniZinc (like most other modelling languages) allows the use of data files to set the value of parameters declared in the original model. This allows the same model to be easily used with different data by running it with different data files.

Data files must have the file extension ".dzn" to indicate a MiniZinc data file and a model can be run with any number of data files (though a variable/parameter can only be assigned a value in one file.

Our new model is shown in Figure 4. We can run it using the command

```
$ mzn-g12fd cakes2.mzn pantry.dzn
```

CAKES2 ≡ [DOWNLOAD]

```
% Baking cakes for the school fete (with data file)

int: flour;  %no. grams of flour available
int: banana; %no. of bananas available
int: sugar;  %no. grams of sugar available
int: butter; %no. grams of butter available
int: cocoa;  %no. grams of cocoa available

constraint assert(flour >= 0,"Invalid datafile: " ++
                  "Amount of flour should be non-negative");
constraint assert(banana >= 0,"Invalid datafile: " ++
                  "Amount of banana should be non-negative");
constraint assert(sugar >= 0,"Invalid datafile: " ++
                  "Amount of sugar should be non-negative");
constraint assert(butter >= 0,"Invalid datafile: " ++
                  "Amount of butter should be non-negative");
constraint assert(cocoa >= 0,"Invalid datafile: " ++
                  "Amount of cocoa should be non-negative");

var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes

% flour
constraint 250*b + 200*c <= flour;
% bananas
constraint 2*b  <= banana;
% sugar
constraint 75*b + 150*c <= sugar;
% butter
constraint 100*b + 150*c <= butter;
% cocoa
constraint 75*c <= cocoa;

% maximize our profit
solve maximize 400*b + 450*c;

output ["no. of banana cakes = \(b)\n",
        "no. of chocolate cakes = \(c)\n"];
```

Figure 4: Data-independent model for determining how many banana and chocolate cakes to bake for the school fete.

PANTRY ≡ [DOWNLOAD]

```
flour = 4000;
banana = 6;
sugar = 2000;
butter = 500;
cocoa = 500;
```

PANTRY2 ≡ [DOWNLOAD]

```
flour = 8000;
banana = 11;
sugar = 3000;
butter = 1500;
cocoa = 800;
```

Figure 5: Example data files for `cakes2.mzn`

where the data file `pantry.dzn` is defined in Figure 5 gives the same result as `cakes.mzn`. The output from running the command

```
$ mzn-g12fd  cakes2.mzn pantry2.dzn
```

with an alternate data set defined in Figure 5 the output is

```
no. of banana cakes = 3
no. of chocolate cakes = 8
----------
==========
```

If we remove the output statement from `cakes.mzn` then MiniZinc will use a default output. In this case the resulting output will be

```
b = 3;
c = 8;
----------
==========
```

Default Output

A MiniZinc model with no output will output a line for each decision variable with its value, unless it is assigned an expression on its declaration. Note how the output is in the form of a correct datafile.

Small data files can be entered without directly creating a `.dzn` file, using the command line flag `-D` *string*, where *string* is the contents of the data file. For example the command

```
$ mzn-g12fd cakes2.mzn -D \
     "flour=4000;banana=6;sugar=2000;butter=500;cocoa=500;"
```

will give identical results to

```
$ mzn-g12fd cakes2.mzn pantry.dzn
```

Data files can only contain assignment statements for decision variables and parameters in the model(s) for which they are intended.

Defensive programming suggests that we should check that the values in the data file are reasonable. For our example it is sensible to check that the quantity of all ingredients is non-negative and generate a run-time error if this is not true. MiniZinc provides a built-in Boolean operator for checking parameter values. The form is *assert(B,S)*. The Boolean expression B is evaluated and if it is false execution aborts and the string expression S is evaluated and printed as an error message. To check and generate an appropriate error message if the amount of flour is negative we can simply add the line

```
constraint assert(flour >= 0,"Amount of flour is non-negative");
```

to our model. Notice that the *assert* expression is a Boolean expression and so is regarded as a type of constraint. We can add similar lines to check that the quantity of the other ingredients is non-negative.

2.4 Real Number Solving

MiniZinc also supports "real number" constraint solving using floating point solving. Consider a problem of taking out a short loan for one year to be repaid in 4 quarterly instalments. A model for this is shown in Figure 6. It uses a simple interest calculation to calculate the balance after each quarter.

Note that we declare a float variable f similar to an integer variable using the keyword `float` instead of `int`.

Float Variable Declarations

A float parameter variable is declared as either:

`float` : ⟨*var-name*⟩
⟨*l*⟩ `..` ⟨*u*⟩ : ⟨*var-name*⟩

where l and u are fixed floating point expressions.
A float decision variable is declared as either:

`var float` : ⟨*var-name*⟩
`var` ⟨*l*⟩ `..` ⟨*u*⟩ : ⟨*var-name*⟩

where l and u are fixed floating point expressions.

We can use the same model to answer a number of different questions. The first question is: if I borrow $1000 at 4% and repay $260 per quarter, how much do I end up owing? This question is encoded by the data file `loan1.dzn`.

Since we wish to use real number solving we need to use a different solver than the finite domain solver used by `mzn-g12fd`. A suitable solver would be one that supports mixed integer linear programming. The MiniZinc distribution contains such a solver. We can invoke it using the command `mzn-g12mip`

LOAN ≡ [DOWNLOAD]

```
% variables
var float: R;        % quarterly repayment
var float: P;        % principal initially borrowed
var 0.0 .. 10.0: I;  % interest rate

% intermediate variables
var float: B1; % balance after one quarter
var float: B2; % balance after two quarters
var float: B3; % balance after three quarters
var float: B4; % balance owing at end

constraint B1 = P * (1.0 + I) - R;
constraint B2 = B1 * (1.0 + I) - R;
constraint B3 = B2 * (1.0 + I) - R;
constraint B4 = B3 * (1.0 + I) - R;

solve satisfy;

output [
 "Borrowing ", show_float(0, 2, P), " at ", show(I*100.0),
 "% interest, and repaying ", show_float(0, 2, R),
  "\nper quarter for 1 year leaves ", show_float(0, 2, B4), " owing\n"
];
```

Figure 6: Model for determining relationships between a 1 year loan repaying every quarter.

```
$ mzn-g12mip loan.mzn loan1.dzn
```

The output is

```
Borrowing 1000.00 at 4.0% interest, and repaying 260.00
per quarter for 1 year leaves 65.78 owing
----------
```

The second question is if I want to borrow $1000 at 4% and owe nothing at the end, how much do I need to repay? This question is encoded by the data file `loan2.dzn`. The output from running the command

```
$ mzn-g12mip loan.mzn loan2.dzn
```

is

LOAN1 ≡ [DOWNLOAD]

```
I = 0.04;
P = 1000.0;
R = 260.0;
```

LOAN2 ≡ [DOWNLOAD]

```
I = 0.04;
P = 1000.0;
B4 = 0.0;
```

LOAN3 ≡ [DOWNLOAD]

```
I = 0.04;
R = 250.0;
B4 = 0.0;
```

Figure 7: Example data files for `loan.mzn`

```
Borrowing 1000.00 at 4.0% interest, and repaying 275.49
per quarter for 1 year leaves 0.00 owing
----------
```

The third question is if I can repay $250 a quarter, how much can I borrow at 4% to end up owing nothing? This question is encoded by the data file `loan3.dzn`. The output from running the command

```
$ mzn-g12mip loan.mzn loan3.dzn
```

is

```
Borrowing 907.47 at 4.0% interest, and repaying 250.00
per quarter for 1 year leaves 0.00 owing
----------
```

Float Arithmetic Operators

MiniZinc provides the standard floating point arithmetic operators: addition (`+`), subtraction (`-`), multiplication (`*`) and floating point division (`/`). It also provides `+` and `-` as unary operators.

MiniZinc does not automatically coerce integers to floating point numbers. The built-in function `int2float` can be used for this purpose.

MiniZinc provides in addition the following floating point functions: absolute value (`abs`), square root (`sqrt`), natural logarithm (`ln`), logarithm base 2 (`log2`), logarithm base 10 (`log10`), exponentiation of e (`exp`), sine (`sin`), cosine (`cos`), tangent (`tan`), arcsine (`asin`), arccosine (`acos`), arctangent (`atan`), hyperbolic sine (`sinh`), hyperbolic cosine (`cosh`), hyperbolic tangent (`tanh`), hyperbolic arcsine (`asinh`), hyperbolic arccosine (`acosh`), hyperbolic arctangent (`atanh`), and power (`pow`) which is the only binary function, the rest are unary.

The syntax for arithmetic literals is reasonably standard. Example float literals are `1.05`, `1.3e-5` and `1.3E+5`.

2.5 Basic structure of a model

We are now in a position to summarise the basic structure of a MiniZinc model. It consists of multiple items each of which has a semicolon ';' at its end. Items can occur in any order. For example, identifiers need not be declared before they are used.

There are 8 kinds of items.

- Include items allow the contents of another file to be inserted into the model. They have the form:

  include ⟨*filename*⟩;

  where *filename* is a string literal. They allow large models to be split into smaller sub-models and also the inclusion of constraints defined in library files. We shall see an example in Figure 11.

- Variable declarations declare new variables. Such variables are global variables and can be referred to from anywhere in the model. Variables come in two kinds. Parameters which are assigned a fixed value in the model or in a data file and decision variables whose value is found only when the model is solved. We say that parameters are *fixed* and decision variables *unfixed*. The variable can be optionally assigned a value as part of the declaration. The form is:

  ⟨*type inst expr*⟩: ⟨*variable*⟩ [= ⟨*expression*⟩];

  The *type-inst expr* gives the instantiation and type of the variable. These are one of the more complex aspects of MiniZinc. Instantiations are declared using `par` for parameters and `var` for decision variables. If there is no explicit instantiation declaration then the variable is a parameter. The type can be a base type, an integer or float range or an array or a set. The base types are `float`, `int`, `string`, `bool`, `ann` of which only `float`, `int` and `bool` can be used for decision variables. The base type `ann` is an annotation—we shall discuss annotations in section 6. Integer range expressions can be used instead of the type `int`. Similarly float range expressions can be used instead of type `float`. These are typically used to give the domain of an integer decision variable but can also be used to restrict the range of an integer parameter. Another use of variable declarations is to define enumerated types—which we discuss in subsection 3.4.

- Assignment items assign a value to a variable. They have the form:

  ⟨*variable*⟩ = ⟨*expression*⟩;

  Values can be assigned to decision variables in which case the assignment is equivalent to writing `constraint` ⟨*variable*⟩ = ⟨*expression*⟩;

- Constraint items form the heart of the model. They have the form:

  constraint ⟨*Boolean expression*⟩;

We have already seen examples of simple constraints using arithmetic comparison and the built-in *assert* operator. In the next section we shall see examples of more complex constraints.

- Solve items specify exactly what kind of solution is being looked for. As we have seen they have one of three forms:

```
solve satisfy;
solve maximize ⟨arithmetic expression⟩;
solve minimize ⟨arithmetic expression⟩;
```

A model is required to have exactly one solve item.

- Output items are for nicely presenting the results of the model execution. They have the form:

```
output [ ⟨string expression⟩, ··· , ⟨string expression⟩ ];
```

If there is no output item, MiniZinc will by default print out the values of all the decision variables which are not optionally assigned a value in the format of assignment items.

- Enumerated type declarations. We discuss these in subsection 3.1 and subsection 3.4.
- Predicate function and test items are for defining new constraints, functions and Boolean tests. We discuss these in section 4.
- The annotation item is used to define a new annotation. We discuss these in section 6.

3 More Complex Models

In the last section we introduced the basic structure of a MiniZinc model. In this section we introduce the array and set data structures, enumerated types and more complex constraints.

3.1 Arrays and Sets

Almost always we are interested in building models where the number of constraints and variables is dependent on the input data. In order to do so we will usually use arrays.

Consider a simple finite element model for modelling temperatures on a rectangular sheet of metal. We approximate the temperatures across the sheet by breaking the sheet into a finite number of elements in a 2 dimensional matrix. A model is shown in Figure 8. It declares the width `w` and height `h` of the finite element model. The declaration

ARRAYDEC ≡
```
set of int: HEIGHT = 0..h;
set of int: CHEIGHT = 1..h-1;
set of int: WIDTH = 0..w;
set of int: CWIDTH = 1..w-1;
array[HEIGHT,WIDTH] of var float: t; % temperature at point (i,j)
```

declares four fixed sets of integers describing the dimensions of the finite element model: `HEIGHT` is the whole height of the model, while `CHEIGHT` is the centre of the height omitting the top and bottom, `WIDTH` is the whole width of the model, while `CWIDTH` is the centre of the width omitting the left and rightsides, Finally a two dimensional array of float variables `t` with rows numbered 0 to h (`HEIGHT`) and columns 0 to h (`WIDTH`), to represent the temperatures at each point in the metal plate. We can access the element of the array in the i^{th} row and j^{th} column using an expression `t[` i `,` j `]`.

Laplace's equation states that when the plate reaches a steady state the temperature at each internal point is the average of its orthogonal neighbours. The constraint

EQUATION ≡
```
% Laplace equation: each internal temp. is average of its neighbours
constraint forall(i in CHEIGHT, j in CWIDTH)(
              4.0*t[i,j] = t[i-1,j] + t[i,j-1] + t[i+1,j] + t[i,j+1]);
```

ensures each internal point (i, j) to the average of its four orthogonal neighbours. The constraints

SIDES ≡
```
% edge constraints
constraint forall(i in CHEIGHT)(t[i,0] = left);
constraint forall(i in CHEIGHT)(t[i,w] = right);
constraint forall(j in CWIDTH)(t[0,j] = top);
constraint forall(j in CWIDTH)(t[h,j] = bottom);
```

LAPLACE ≡ [DOWNLOAD]

```
int: w = 4;
int: h = 4;

▶ ARRAYDEC
var float: left;   % left   edge temperature
var float: right;  % right  edge temperature
var float: top;    % top    edge temperature
var float: bottom; % bottom edge temperature

▶ EQUATION
▶ SIDES
▶ CORNERS
left = 0.0;
right = 0.0;
top = 100.0;
bottom = 0.0;

solve satisfy;

output [ show_float(6, 2, t[i,j]) ++
         if j == h then "\n" else " " endif |
         i in HEIGHT, j in WIDTH
];
```

Figure 8: Finite element plate model for determining steady state temperatures (`laplace.mzn`).

constrains the temperatures on each edge to be equal, and gives these temperatures names: `left`, `right`, `top` and `bottom`. While the constraints

CORNERS ≡

```
% corner constraints
constraint t[0,0]=0.0;
constraint t[0,w]=0.0;
constraint t[h,0]=0.0;
constraint t[h,w]=0.0;
```

ensure that the corners (which are irrelevant) are set to 0.0. We can determine the temperatures in a plate broken into 5 × 5 elements with left, right and bottom temperature 0 and top temperature 100 with the model shown in Figure 8.

Running the command

```
$ mzn-g12mip laplace.mzn
```

gives the output

```
    0.00 100.00 100.00 100.00   0.00
    0.00  42.86  52.68  42.86   0.00
    0.00  18.75  25.00  18.75   0.00
    0.00   7.14   9.82   7.14   0.00
    0.00   0.00   0.00   0.00   0.00
  ----------
```

Sets

Set variables are declared with a declaration of the form

`set of` ⟨*type-inst*⟩ `:` ⟨*var-name*⟩ `;`

where sets of integers, enums (see later), floats or Booleans are allowed. The only type allowed for decision variable sets are variables sets of integers or enums. Set literals are of form

`{` ⟨$expr_1$⟩, ... , ⟨$expr_n$⟩ `}`

or are range expressions over either integers, enums or floats of form

⟨$expr_1$⟩ `..` ⟨$expr_2$⟩

The standard set operations are provided: element membership (`in`), (non-strict) subset relationship (`subset`), (non-strict) superset relationship (`superset`), union (`union`), intersection (`intersect`), set difference (`diff`), symmetric set difference (`symdiff`) and the number of elements in the set (`card`).
As we have seen set variables and set literals (including ranges) can be used as an implicit type in variable declarations in which case the variable has the type of the elements in the set and the variable is implicitly constrained to be a member of the set.

Our cake baking problem is an example of a very simple kind of production planning problem. In this kind of problem we wish to determine how much of each kind of product to make to maximise the profit where manufacturing a product consumes varying amounts of some fixed resources. We can generalise the MiniZinc model in Figure 4 to handle this kind of problem with a model that is generic in the kinds of resources and products. The model is shown in Figure 9 and a sample data file (for the cake baking example) is shown in Figure 10.

The new feature in this model is the use of enumerated types. These allow us to treat the choice of resources and products as parameters to the model. The first item in the model

```
enum Products;
```

declares `Products` as an *unknown* set of products.

SIMPLE-PROD-PLANNING ≡ [DOWNLOAD]

```
% Products to be produced
enum Products;
% profit per unit for each product
array[Products] of int: profit;
% Resources to be used
enum Resources;
% amount of each resource available
array[Resources] of int: capacity;

% units of each resource required to produce 1 unit of product
array[Products, Resources] of int: consumption;
constraint assert(forall (r in Resources, p in Products)
           (consumption[p,r] >= 0), "Error: negative consumption");

% bound on number of Products
int: mproducts = max (p in Products)
                     (min (r in Resources where consumption[p,r] > 0)
                          (capacity[r] div consumption[p,r]));

% Variables: how much should we make of each product
array[Products] of var 0..mproducts: produce;
array[Resources] of var 0..max(capacity): used;

% Production cannot use more than the available Resources:
constraint forall (r in Resources) (
      used[r] = sum (p in Products)(consumption[p, r] * produce[p])
)
constraint forall (r in Resources) (
      used[r] <= capacity[r]
);

% Maximize profit
solve maximize sum (p in Products) (profit[p]*produce[p]);

output [ "\(p) = \(produce[p]);\n" | p in Products ] ++
       [ "\(r) = \(used[r]);\n" | r in Resources ];
```

Figure 9: Model for simple production planning (`simple-prod-planning.mzn`)

.

SIMPLE-PROD-PLANNING-DATA ≡ [DOWNLOAD]

```
% Data file for simple production planning model
Products = { BananaCake, ChocolateCake };
profit = [400, 450]; % in cents

Resources = { Flour, Banana, Sugar, Butter, Cocoa };
capacity = [4000, 6, 2000, 500, 500];

consumption= [| 250, 2, 75, 100, 0,
              | 200, 0, 150, 150, 75 |];
```

Figure 10: Example data file for the simple production planning problem.

Enumerated Types

Enumerated types, which we shall refer to as `enums`, are declared with a declaration of the form

enum ⟨*var-name*⟩;

An enumerated type is defined by an assignment of the form

⟨*var-name*⟩ = { ⟨*var-name*$_1$⟩, ... , ⟨*var-name*$_n$⟩ } ;

where *var-name*$_1$, ..., *var-name*$_n$ are the elements of the enumerated type, with name *var-name*. Each of the elements of the enumerated type is also effectively declared by this definition as a new constant of that type. The declaration and definition can be combined into one line as usual.

The second item declares an array of integers:

```
array[Products] of int: profit;
```

The index set of the array `profit` is `Products`. Ideally this would mean that only elements of the set `Products` could be used to index the array. But enumerated types in MiniZinc are treated similar to integers so at present the only guarantee is that only $1, 2, ..., |Products|$ are valid indices into the array. The array access `profit[i]` gives the profit for product `i`.

The elements of an enumerated type of n elements act very similar to the integers $1..n$. They can be compared, they are ordered, by the order they appear in the enumerated type definition, they can be interated over, they can appear as indices of arrays, in fact they can appear almost anywhere an integer can appear.

In the example data file we have initialized the array using a list of integers

```
Products = { BananaCake, ChocolateCake };
profit = [400,450];
```

meaning the profit for a banana cake is 400, while for a chocolate cake it is 450. Internally `BananaCake` will be treated like the integer 1, while `ChocolateCake` will be treated like the integer 2. While MiniZinc does not provide an explicit list type, one-dimensional arrays with an index set $1..n$ behave like lists, and we will sometimes refer to them as lists.

In a similar fashion, in the next 2 items we declare a set of resources `Resources`, and an array `capacity` which gives the amount of each resource that is available.

More interestingly, the item

```
array[Products, Resources] of int: consumption;
```

declares a 2-D array `consumption`. The value of `consumption[p,r]` is the amount of resource `r` required to produce one unit of product `p`. Note that the first index is the row and the second is the column.

The data file contains an example initialization of a 2-D array:

```
consumption= [| 250, 2, 75,  100, 0,
              | 200, 0, 150, 150, 75 |];
```

Notice how the delimiter | is used to separate rows.

Arrays

Thus, MiniZinc provides one- and multi-dimensional arrays which are declared using the type:

array[⟨*index-set*$_1$⟩, ..., ⟨*index-set*$_n$⟩] **of** ⟨*type-inst*⟩

MiniZinc requires that the array declaration contains the index set of each dimension and that the index set is either an integer range, a set variable initialised to an integer range, or an enumeration type. Arrays can contain any of the base types: integers, enums, Booleans, floats or strings. These can be fixed or unfixed except for strings which can only be parameters. Arrays can also contain sets but they cannot contain arrays.

One-dimensional array literals are of form

[⟨*expr*$_1$⟩, ... , ⟨*expr*$_n$⟩]

while two-dimensional array literals are of form

[| ⟨*expr*$_{1,1}$⟩, ... , ⟨*expr*$_{1,n}$⟩, | ..., | ⟨*expr*$_{m,1}$⟩, ... , ⟨*expr*$_{m,n}$⟩ |]

where the array has m rows and n columns.

The family of built-in functions `array1d`, `array2d`, etc, can be used to initialise an array of any dimension from a list (or more exactly a one-dimensional array). The call:

array$n$d(⟨*index-set*$_1$⟩, ..., ⟨*index-set*$_n$⟩, ⟨*list*⟩)

returns an n dimensional array with index sets given by the first n arguments and the last argument contains the elements of the array. For instance, **`array2d`**`(1..3, 1..2, [1, 2, 3, 4, 5, 6])` is equivalent to `[|1, 2 |3, 4 |5, 6|])`.

Array elements are accessed in the usual way: `a[i,j]` gives the element in the i^{th} row and j^{th} column.

The concatenation operator '++' can be used to concatenate two one-dimensional arrays together. The result is a list, i.e. a one-dimensional array whose elements are indexed from 1. For instance `[4000, 6] ++ [2000, 500, 500]` evaluates to `[4000, 6, 2000, 500, 500]`. The built-in function `length` returns the length of a one-dimensional array.

The next item in the model defines the parameter `mproducts`. This is set to an upper-bound on the number of products of any type that can be produced. This is quite a complex example of nested array comprehensions and aggregation operators. We shall introduce these before we try to understand this item and the rest of the model.

First, MiniZinc provides list comprehensions similar to those provided in many functional programming languages. For example, the list comprehension `[i + j | i, j in 1..3 where j < i]` evaluates to `[1 + 2, 1 + 3, 2 + 3]` which is `[3, 4, 5]`. Of course `[3, 4, 5]` is simply an array with index set 1..3.

MiniZinc also provides set comprehensions which have a similar syntax: for instance, `{i + j | i, j in 1..3 where j < i}` evaluates to the set `{3, 4, 5}`.

List and Set Comprehensions

The generic form of a list comprehension is

[⟨*expr*⟩ | ⟨*generator-exp*⟩]

The expression ⟨*expr*⟩ specifies how to construct elements in the output list from the elements generated by ⟨*generator-exp*⟩. The generator ⟨*generator-exp*⟩ consists of a comma separated sequence of generator expressions optionally followed by a Boolean expression. The two forms are

⟨*generator*⟩, ..., ⟨*generator*⟩
⟨*generator*⟩, ..., ⟨*generator*⟩ **where** ⟨*bool-exp*⟩

The optional ⟨*bool-exp*⟩ in the second form acts as a filter on the generator expression: only elements satisfying the Boolean expression are used to construct elements in the output list. A generator ⟨*generator*⟩ has form

⟨*identifier*⟩,..., ⟨*identifier*⟩ **in** ⟨*array-exp*⟩

Each identifier is an *iterator* which takes the values of the array expression in turn, with the last identifier varying most rapidly.
The generators of a list comprehension, and ⟨*bool-exp*⟩ usually do not involve decision variables. If they do involve decision variables then the list produced is a list of `var opt` *T* where *T* is the type of the ⟨*expr*⟩. See the discussion of option types in section 5 for more details.
Set comprehensions are almost identical to list comprehensions: the only difference is the use of '{' and '}' to enclose the expression rather than '[' and ']'. The elements generated by a set comprehension must be fixed, i.e. free of decision variables. Similarly the generators and optional ⟨*bool-exp*⟩ for set comprehensions must be fixed.

Second, MiniZinc provides a number of built-in functions that take a one-dimensional array and aggregate the elements. Probably the most useful of these is `forall`. This takes an array of Boolean expressions (that is, constraints) and returns a single Boolean expression which is the logical conjunction of the Boolean expressions in the array.

For example, consider the expression

```
forall( [a[i] != a[j] | i,j in 1..3 where i < j])
```

where `a` is an arithmetic array with index set 1..3. This constrains the elements in `a` to be different. The list comprehension evaluates to `[ a[1] != a[2], a[1] != a[3], a[2] != a[3] ]` and so the `forall` function returns the logical conjunction `a[1] != a[2]` ∧ `a[1] != a[3]` ∧ `a[2] != a[3]`.

Aggregation functions

The *aggregation functions* for arithmetic arrays are: `sum` which adds the elements, `product` which multiplies them together, and `min` and `max` which respectively return the least and greatest element in the array. When applied to an empty array, `min` and `max` give a run-time error, `sum` returns 0 and `product` returns 1.
MiniZinc provides four aggregation functions for arrays containing Boolean expressions. As we have seen, the first of these, `forall`, returns a single constraint which is the logical conjunction of the constraints. The second function, `exists`, returns the logical disjunction of the constraints. Thus, `forall` enforces that all constraints in the array hold, while `exists` ensures that at least one of the constraints holds. The third function, `xorall`, ensures that an odd number of constraints hold. The fourth function, `iffall`, ensures that an even number of constraints holds.

The third, and final, piece in the puzzle is that MiniZinc allows a special syntax for aggregation functions when used with an array comprehension. Instead of writing

```
forall( [a[i] != a[j] | i,j in 1..3 where i < j])
```

the modeller can instead write the more mathematical looking

```
forall (i,j in 1..3 where i < j) (a[i] != a[j])
```

The two expressions are completely equivalent: the modeller is free to use whichever they feel looks most natural.

Generator call expressions

A *generator call expression* has form

⟨*agg-func*⟩ `(` ⟨*generator-exp*⟩ `)` `(` ⟨*exp*⟩ `)`

The round brackets around the generator expression ⟨*generator-exp*⟩ and the constructor expression ⟨*exp*⟩ are not optional: they must be there. This is equivalent to writing

⟨*agg-func*⟩`(` `[` ⟨*expr*⟩ `|` ⟨*generator-exp*⟩ `]` `)`

The aggregation function ⟨*agg-func*⟩ is any MiniZinc functions expecting a single array as argument.

We are now in a position to understand the rest of the simple production planning model shown in Figure 9. For the moment ignore the item defining `mproducts`. The item afterwards:

```
array[Products] of var 0..mproducts: produce;
```

defines a one-dimensional array `produce` of decision variables. The value of `produce[p]` will be set to the amount of product `p` in the optimal solution. The next item

```
array[Resources] of var 0..max(capacity): used;
```

defines a set of auxiliary variables that record how much of each resource is used. The next two constraints

```
constraint forall (r in Resources)
           (used[r] = sum (p in Products) (consumption[p, r] * produce[p]));
constraint forall (r in Resources)(used[r] <= capacity[r] );
```

computes in `used[r]` the total consumption of the resource `r` and ensures it is less than the available amount. Finally, the item

```
solve maximize sum (p in Products) (profit[p]*produce[p]);
```

indicates that this is a maximisation problem and that the objective to be maximised is the total profit.

We now return to the definition of `mproducts`. For each product `p` the expression

```
(min (r in Resources where consumption[p,r] > 0)
                                (capacity[r] div consumption[p,r])
```

determines the maximum amount of `p` that can be produced taking into account the amount of each resource `r` and how much of `r` is required to produce the product. Notice the use of the filter `where consumption[p,r] > 0` to ensure that only resources required to make the product are considered so as to avoid a division by zero error. Thus, the complete expression

```
int: mproducts = max (p in Products)
                     (min (r in Resources where consumption[p,r] > 0)
                                (capacity[r] div consumption[p,r]));
```

computes the maximum amount of *any* product that can be produced, and so this can be used as an upper bound on the domain of the decision variables in `produce`.

Finally notice the output item is more complex, and uses list comprehensions to create an understandable output. Running

```
$ mzn-g12fd simple-prod-planning.mzn simple-prod-planning-data.dzn
```

results in the output

```
BananaCake = 2;
ChocolateCake = 2;
Flour = 900;
Banana = 4;
Sugar = 450;
Butter = 500;
Cocoa = 150;
----------
```

SEND-MORE-MONEY ≡ [DOWNLOAD]

```
include "alldifferent.mzn";

var 1..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 1..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;

constraint           1000 * S + 100 * E + 10 * N + D
                   + 1000 * M + 100 * O + 10 * R + E
       = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;

constraint alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;

output ["   \(S)\(E)\(N)\(D)\n",
        "+  \(M)\(O)\(R)\(E)\n",
        "= \(M)\(O)\(N)\(E)\(Y)\n"];
```

Figure 11: Model for the cryptarithmetic problem SEND+MORE=MONEY (`send-more-money.mzn`).

3.2 Global Constraints

MiniZinc includes a library of global constraints which can also be used to define models. An example is the `alldifferent` constraint which requires all the variables appearing in its argument to be different.

The SEND+MORE=MONEY problem requires assigning a different digit to each letter so that the arithmetic constraint holds. The model shown in Figure 11 uses the constraint expression `alldifferent([S,E,N,D,M,O,R,Y])` to ensure that each letter takes a different digit value. The global constraint is made available in the model using include item

```
include "alldifferent.mzn";
```

which makes the global constraint `alldifferent` usable by the model. One could replace this line by

```
include "globals.mzn";
```

which includes all globals.

A list of all the global constraints defined for MiniZinc is included in the release documentation. See subsection 4.1 for a description of some important global constraints.

3.3 Conditional Expressions

MiniZinc provides a conditional *if-then-else-endif* expression. An example of its use is

```
int: r = if y != 0 then x div y else 0 endif;
```

which sets r to x divided by y unless y is zero in which case it sets it to zero.

Conditional expressions

The form of a conditional expression is

if ⟨*boolexp*⟩ **then** ⟨exp_1⟩ **else** ⟨exp_2⟩ **endif**

It is a true expression rather than a control flow statement and so can be used in other expressions. It evaluates to ⟨exp_1⟩ if the Boolean expression ⟨*boolexp*⟩ is true and ⟨exp_2⟩ otherwise. The type of the conditional expression is that of ⟨exp_1⟩ and ⟨exp_2⟩ which must have the same type.
If the ⟨*boolexp*⟩ contains decision variables, then the type-inst of the expression is `var` T where T is the type of ⟨exp_1⟩ and ⟨exp_2⟩ even if both expressions are fixed.

Conditional expressions are very useful in building complex models, or complex output. Consider the model of Sudoku problems shown in Figure 12. The initial board positions are given by the `start` parameter where 0 represents an empty board position. This is converted to constraints on the decision variables `puzzle` using the conditional expression

```
constraint forall(i,j in PuzzleRange)(
     if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );
```

Conditional expressions are also very useful for defining complex output. In the Sudoku model of Figure 12 the expression

```
if j mod S == 0 then " " else "" endif
```

inserts an extra space between groups of size S. The output expression also use conditional expressions to and add blank lines after each S lines. The resulting output is highly readable.

The remaining constraints ensure that the numbers appearing in each row and column and $S \times S$ subsquare are all different.

One can use MiniZinc to search for all solutions to a satisfaction problem (`solve satisfy`) by using the flag `-a` or `-all-solutions`. Running

```
$ mzn-g12fd --all-solutions sudoku.mzn sudoku.dzn
```

results in

SUDOKU ≡ [DOWNLOAD]

```
include "alldifferent.mzn";

int: S;
int: N = S * S;
int: digs = ceil(log(10.0,int2float(N))); % digits for output

set of int: PuzzleRange = 1..N;
set of int: SubSquareRange = 1..S;

array[1..N,1..N] of 0..N: start; %% initial board 0 = empty
array[1..N,1..N] of var PuzzleRange: puzzle;

% fill initial board
constraint forall(i,j in PuzzleRange)(
    if start[i,j] > 0 then puzzle[i,j] = start[i,j] else true endif );

% All different in rows
constraint forall (i in PuzzleRange) (
                  alldifferent( [ puzzle[i,j] | j in PuzzleRange ]) );

% All different in columns.
constraint forall (j in PuzzleRange) (
                  alldifferent( [ puzzle[i,j] | i in PuzzleRange ]) );

% All different in sub-squares:
constraint
        forall (a, o in SubSquareRange)(
                alldifferent( [ puzzle[(a-1) *S + a1, (o-1)*S + o1] |
                                        a1, o1 in SubSquareRange ] ) );

solve satisfy;

output  [ show_int(digs,puzzle[i,j]) ++ " " ++
         if j mod S == 0 then " " else "" endif ++
         if j == N then
             if i != N then
                if i mod S == 0 then "\n\n" else "\n" endif
         else "" endif else "" endif
         | i,j in PuzzleRange ] ++ ["\n"];
```

Figure 12: Model for generalized Sudoku problem (`sudoku.mzn`).

SUDOKU.DZN ≡ [DOWNLOAD]

```
S=3;
start=[|
0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 6, 8, 4, 0, 1, 0, 7, 0|
0, 0, 0, 0, 8, 5, 0, 3, 0|
0, 2, 6, 8, 0, 9, 0, 4, 0|
0, 0, 7, 0, 0, 0, 9, 0, 0|
0, 5, 0, 1, 0, 6, 3, 2, 0|
0, 4, 0, 6, 1, 0, 0, 0, 0|
0, 3, 0, 2, 0, 7, 6, 9, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0|];
```

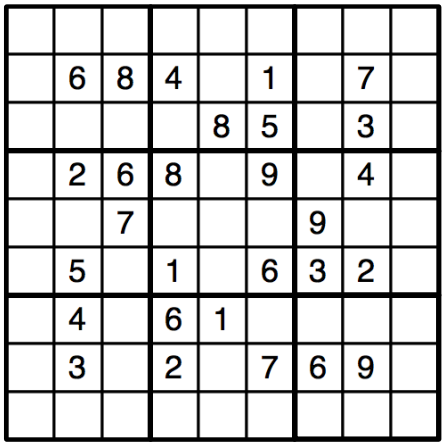


Figure 13: Example data file for generalised Sudoku problem (`sudoku.dzn`) and the problem it represents.

```
 5 9 3  7 6 2  8 1 4
 2 6 8  4 3 1  5 7 9
 7 1 4  9 8 5  2 3 6

 3 2 6  8 5 9  1 4 7
 1 8 7  3 2 4  9 6 5
 4 5 9  1 7 6  3 2 8

 9 4 2  6 1 8  7 5 3
 8 3 5  2 4 7  6 9 1
 6 7 1  5 9 3  4 8 2
----------
==========
```

The line `==========` is output when the system has output all possible solutions, here verifying that there is exactly one.

3.4 Enumerated Types

Enumerated types allows us to build models that depend on a set of objects which are part of the data, or are named in the model, and hence make models easier to understand and debug. We have

AUST-ENUM ☰ [DOWNLOAD]

```
enum Color;
var Color: wa;
var Color: nt;
var Color: sa;
var Color: q;
var Color: nsw;
var Color: v;
var Color: t;
constraint wa != nt /\ wa != sa /\ nt != sa /\ nt != q /\ sa != q;
constraint sa != nsw /\ sa != v /\ q != nsw /\ nsw != v;
solve satisfy;
```

Figure 14: Model for coloring Australia using enumerated types (`aust-enum.mzn`).

introduce enumerated types or enums briefly, in this subsection we will explore how we can use them more fully, and show some of the built in functions for dealing with enumerated types.

Let's revisit the problem of coloring the graph of Australia from section 2.

The model shown in Figure 14 declares an enumerated type `Color` which must be defined in the data file. Each of the state variables is declared to take a value from this enumerated type. Running this program using

```
$ minizinc -D"Color = { red, yellow, blue };" aust-enum.mzn
```

might result in output

```
wa = yellow;
nt = blue;
sa = red;
q = yellow;
nsw = blue;
v = yellow;
t = red;
```

Enumerated Type Variable Declarations

An enumerated type parameter variable is declared as either:

⟨*enum-name*⟩ : ⟨*var-name*⟩
⟨*l*⟩ .. ⟨*u*⟩ : ⟨*var-name*⟩

where *enum-name* is the name of a enumerated type, and l and u are fixed enumerated type expressions of the same enumerated type.
An enumerated type decision variable is declared as either:

var ⟨*enum-name*⟩ : ⟨*var-name*⟩
var ⟨*l*⟩ .. ⟨*u*⟩ : ⟨*var-name*⟩

where *enum-name* is the name of a enumerated type, and l and u are fixed enumerated type expressions of the same enumerated type.

A key behaviour of enumerated types is that they are automatically coerced to integers when they are used in a position expecting an integer. For example, this allows us to use global constraints defined on integers, e.g.

```
global_cardinality_low_up([wa,nt,sa,q,nsw,v,t],
                          [red,yellow,blue],[2,2,2],[2,2,3]);
```

requires at least two states to be colored each color and three to be colored blue.

Enumerated Type Operations

There are a number of built in operations on enumerated types:

- `enum_next`(X,x): returns the next value in after x in the enumerated type X. This is a partial function, if x is the last value in the enumerated type X then the function returns $\bot$ causing the Boolean expression containing the expression to evaluate to *false*.
- `enum_prev`(X,x): returns the previous value before x in the enumerated type X. Similarly `enum_prev` is a partial function.
- `to_enum`(*Enum*,i): maps an integer expression i to an enumerated type value in type *Enum* or evaluates to $\bot$ if i is less than or equal to 0 or greater then the number of elements in *Enum*.

Note also that a number of standard functions are applicable to enumerated types

- `card`(*Enum*): returns the cardinality of an enumerated type *Enum*.
- `min`(*Enum*): returns the minimum element of of an enumerated type *Enum*.
- `max`(*Enum*): returns the maximum element of of an enumerated type *Enum*.

3.5 Complex Constraints

Constraints are the core of the MiniZinc model. We have seen simple relational expressions but constraints can be considerably more powerful than this. A constraint is allowed to be any Boolean expression. Imagine a scheduling problem in which we have two tasks that cannot overlap in time. If `s1` and `s2` are the corresponding start times and `d1` and `d2` are the corresponding durations we can express this as:

```
  constraint s1 + d1 <= s2  \/ s2 + d2 <= s1;
```

which ensures that the tasks do not overlap.

Booleans

Boolean expressions in MiniZinc can be written using a standard mathematical syntax. The Boolean literals are `true` and `false` and the Boolean operators are conjunction, i.e. and (`/\`), disjunction, i.e. or (`\/`), only-if (`<-`), implies (`->`), if-and-only-if (`<->`) and negation (`not`). The built-in function `bool2int` coerces Booleans to integers: it returns 1 if its argument is true and 0 otherwise.

The job shop scheduling model given in Figure 15 gives a realistic example of the use of this disjunctive modelling capability. In job shop scheduling we have a set of jobs, each consisting of a sequence of tasks on separate machines: so task $[i, j]$ is the task in the i^{th} job performed on the j^{th} machine. Each sequence of tasks must be completed in order, and no two tasks on the same machine can overlap in time. Even small instances of this problem can be quite challenging to find optimal solutions.

The command

```
  $ mzn-g12fd --all-solutions jobshop.mzn jdata.dzn
```

solves a small job shop scheduling problem, and illustrates the behaviour of `all-solutions` for optimisation problems. Here the solver outputs each better solutions as it finds it, rather than all possible optimal solutions. The (partial) output from this command is:

```
end = 41
 0  1  5 10 13
 5  8 10 25 26
 1 10 17 21 28
 8 14 21 26 32
 9 16 22 32 40
----------
```

and after quite a few more solutions then finally:

```
end = 31
 0  3  7 12 18
 6  9 19 26 28
 2 11 15 19 24
 1  2  3  4 10
 9 16 26 28 30
----------
end = 30
 1  2  6 11 17
 6 10 15 22 23
 2  6 11 15 25
 0  1  2  3  9
 9 16 22 24 29
----------
==========
```

JOBSHOP ≡ [DOWNLOAD]

```
enum JOB;
enum TASK;
TASK: last = max(TASK);
array [JOB,TASK] of int: d;                    % task durations
int: total = sum(i in JOB, j in TASK)(d[i,j]);% total duration
int: digs = ceil(log(10.0,int2float(total))); % digits for output
array [JOB,TASK] of var 0..total: s;           % start times
var 0..total: end;                             % total end time


constraint %% ensure the tasks occur in sequence
    forall(i in JOB) (
        forall(j in TASK where j < last)
            (s[i,j] + d[i,j] <= s[i,enum_next(TASK,j)]) /\
        s[i,last] + d[i,last] <= end
    );

constraint %% ensure no overlap of tasks
    forall(j in TASK) (
        forall(i,k in JOB where i < k) (
            s[i,j] + d[i,j] <= s[k,j] \/
            s[k,j] + d[k,j] <= s[i,j]
        )
    );

solve minimize end;

output ["end = \(end)\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == last then "\n" else "" endif |
         i in JOB, j in TASK ];
```

Figure 15: Model for job-shop scheduling problems (`jobshop.mzn`).

indicating an optimal solution with end time 30 is finally found, and proved optimal. We can generate all *optimal solutions* by adding a constraint that `end = 30` and changing the solve item to `solve satisfy` and then executing

```
$ mzn-g12fd --all-solutions jobshop.mzn jobshop.dzn
```

For this problem there are very many optimal solutions.

JDATA ≡ [DOWNLOAD]

```
JOB = anon_enum(5);
TASK = anon_enum(5);
d = [| 1, 4, 5, 3, 6
     | 3, 2, 7, 1, 2
     | 4, 4, 4, 4, 4
     | 1, 1, 1, 6, 8
     | 7, 3, 2, 2, 1 |];
```

Figure 16: Data for job-shop scheduling problems (`jdata.dzn`).

STABLE-MARRIAGE ≡ [DOWNLOAD]

```
int: n;

enum Men = anon_enum(n);
enum Women = anon_enum(n);

array[Women, Men] of int: rankWomen;
array[Men, Women] of int: rankMen;

array[Men] of var Women: wife;
array[Women] of var Men: husband;

► ASSIGNMENT
► RANKING
solve satisfy;

output ["wives= \(wife)\nhusbands= \(husband)\n"];
```

Figure 17: Model for stable marriage problem (`stable-marriage.mzn`).

Another powerful modelling feature in MiniZinc is that decision variables can be used for array access. As an example, consider the (old-fashioned) *stable marriage problem*. We have n (straight) women and n (straight) men. Each man has a ranked list of women and vice versa. We want to find a husband/wife for each women/man so that all marriages are *stable* in the sense that:

- whenever m prefers another women o to his wife w, o prefers her husband to m, and
- whenever w prefers another man o to her husband m, o prefers his wife to w.

This can be elegantly modelled in MiniZinc. The model and sample data is shown in Figure 17 and **??**.

STABLE-MARRIAGE.DZN ≡ [DOWNLOAD]

```
n = 5;
rankWomen =
 [| 1, 2, 4, 3, 5,
  | 3, 5, 1, 2, 4,
  | 5, 4, 2, 1, 3,
  | 1, 3, 5, 4, 2,
  | 4, 2, 3, 5, 1 |];

rankMen =
 [| 5, 1, 2, 4, 3,
  | 4, 1, 3, 2, 5,
  | 5, 3, 2, 4, 1,
  | 1, 5, 4, 3, 2,
  | 4, 3, 2, 1, 5 |];
```

Figure 18: Example data file for the stable marriage problem model shown in Figure 17.

The first three items in the model declare the number of men/women and the set of men and women. Here we introduce the use of *anonymous enumerated types*. Both `Men` and `Women` are sets of size n, but we do not wish to mix them up so we use an anonymous enumerated type. This allows MiniZinc to detect modelling errors where we use `Men` for `Women` or vice versa.

The matrices `rankWomen` and `rankMen`, respectively, give the women's ranking of the men and the men's ranking of the women. Thus, the entry `rankWomen[w,m]` gives the ranking by woman `w` of man `m`. The lower the number in the ranking, the more the man or women is preferred.

There are two arrays of decision variables: `wife` and `husband`. These, respectively, contain the wife of each man and the husband of each women.

The first two constraints

ASSIGNMENT ≡

```
constraint forall (m in Men) (husband[wife[m]]=m);
constraint forall (w in Women) (wife[husband[w]]=w);
```

ensure that the assignment of husbands and wives is consistent: w is the wife of m implies m is the husband of w and vice versa. Notice how in `husband[wife[m]]` the index expression `wife[m]` is a decision variable, not a parameter.

The next two constraints are a direct encoding of the stability condition:

RANKING ≡

```
constraint forall (m in Men, o in Women) (
     rankMen[m,o] < rankMen[m,wife[m]] ->
         rankWomen[o,husband[o]] < rankWomen[o,m] );

constraint forall (w in Women, o in Men) (
     rankWomen[w,o] < rankWomen[w,husband[w]] ->
         rankMen[o,wife[o]] < rankMen[o,w] );
```

MAGIC-SERIES ≡ [DOWNLOAD]

```
int: n;
array[0..n-1] of var 0..n: s;

constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));

solve satisfy;

output [ "s = \(s);\n" ] ;
```

Figure 19: Model solving the magic series problem (`magic-series.mzn`).

This natural modelling of the stable marriage problem is made possible by the ability to use decision variables as array indices and to construct constraints using the standard Boolean connectives. The alert reader may be wondering at this stage, what happens if the array index variable takes a value that is outside the index set of the array. MiniZinc treats this as failure: an array access `a[e]` implicitly adds the constraint `e in index_set(a)` to the closest surrounding Boolean context where `index_set(a)` gives the index set of `a`.

Anonymous Enumerated Types

An *anonymous enumerated type* expression is of the form `enum_anon( n )` where n is a fixed integer expression defining the size of the enumerated type.
An anonymous enumerated type is just like any other enumerated type except that we have no names for its elements. When printed out, they are given unique names based on the enumerated type name.

Thus for example, consider the variable declarations

```
array[1..2] of int: a= [2,3];
var 0..2: x;
var 2..3: y;
```

The constraint `a[x] = y` will succeed with $x = 1 \wedge y = 2$ and $x = 2 \wedge y = 3$. And the constraint `not a[x] = y` will succeed with $x = 0 \wedge y = 2$, $x = 0 \wedge y = 3$, $x = 1 \wedge y = 3$ and $x = 2 \wedge y = 2$.

In the case of invalid array accesses by a parameter, the formal semantics of MiniZinc treats this as failure so as to ensure that the treatment of parameters and decision variables is consistent, but a warning is issued since it is almost always an error.

The coercion function `bool2int` can be called with any Boolean expression. This allows the MiniZinc modeller to use so called *higher order constraints*. As a simple example consider the *magic series problem*: find a list of numbers $s = [s_0, \ldots, s_{n-1}]$ such that s_i is the number of occurrences of i in s. An example is $s = [1, 2, 1, 0]$.

A MiniZinc model for this problem is shown in Figure 19. The use of `bool2int` allows us to sum up the number of times the constraint `s[j]=i` is satisfied. Executing the command

```
$ mzn-g12fd --all-solutions magic-series.mzn -D "n=4;"
```

leads to the output

```
s = [1, 2, 1, 0];
----------
s = [2, 0, 2, 0];
----------
==========
```

indicating exactly two solutions to the problem.

Note that MiniZinc will automatically coerce Booleans to integers and integers to floats when required. We could replace the the constraint item in Figure 19 with

```
constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(s[j]=i)));
```

and get identical results, since the Boolean expression `s[j] = i` will be automatically coerced to an integer, effectively by the MiniZinc system automatically adding the missing `bool2int`.

Coercion

In MiniZinc one can *coerce* a Boolean value to an integer value using the `bool2int` function. Similarly one can coerce an integer value to a float value using `int2float`. The instantiation of the coerced value is the same as the argument, e.g. `par bool` is coerced to `par int`, while `var bool` is coerced to `var int`.
MiniZinc automatically coerces Boolean expressions to integer expressions and integer expressions to float expressions, by inserting `bool2int` and `int2float` in the model appropriately. Note that it will also coerce Booleans to floats using two steps.

3.6 Set Constraints

Another powerful modelling feature of MiniZinc is that it allows sets containing integers to be decision variables: this means that when the model is evaluated the solver will find which elements are in the set.

As a simple example, consider the *0/1 knapsack problem*. This is a restricted form of the knapsack problem in which we can either choose to place the item in the knapsack or not. Each item has a weight and a profit and we want to find which choice of items leads to the maximum profit subject to the knapsack not being too full.

It is natural to model this in MiniZinc with a single decision variable: where `ITEM` is the set of possible items. If the arrays `weight[i]` and `profit[i]` respectively give the weight and profit of item `i`, and the maximum weight the knapsack can carry is given by `capacity` then a naural model is given in Figure 20.

Notice that the `var` keyword comes before the `set` declaration indicating that the set itself is the decision variable. This contrasts with an array in which the `var` keyword qualifies the elements in the array rather than the array itself since the basic structure of the array is fixed, i.e. its index set.

As a more complex example of set constraint consider the social golfers problem shown in Figure 21. The aim is to schedule a golf tournament over `weeks` using `groups` × `size` golfers. Each

KNAPSACK ☰ [DOWNLOAD]

```
enum ITEM;
int: capacity;

array[ITEM] of int: profits;
array[ITEM] of int: weights;

var set of ITEM: knapsack;

constraint sum (i in knapsack) (weights[i]) <= capacity;

solve maximize sum (i in knapsack) (profits[i]) ;

output ["knapsack = \(knapsack)\n"];
```

Figure 20: Model for the 0/1 knapsack problem (`knapsack.mzn`).

week we have to schedule `groups` different groups each of size `size`. No two pairs of golfers should ever play in two groups.

The variables in the model are sets of golfers `Sched[i,j]` for the i^{th} week and j^{th} group,

The constraints shown in Figure 22 first enforces an ordering on the first set in each week to remove symmetry in swapping weeks. Next it enforces an ordering on the sets in each week, and makes each set have a cardinality of `size`. It then ensures that each week is a partition of the set of golfers using the global constraint `partition_set`. Finally the last constraint ensures that no two players play in two groups together (since the cardinality of the intersection of any two groups is at most 1).

There are also symmetry breaking initialisation constraints shown in Figure 23: the first week is fixed to have all players in order; the second week is made up of the first players of each of the first groups in the first week; finally the model forces the first `size` players to appear in their corresponding group number for the remaining weeks.

Executing the command

```
$ mzn-g12fd social-golfers.mzn social-golfers.dzn
```

where the data file defines a problem with 4 weeks, with 4 groups of size 3 leads to the output

```
1..3 4..6 7..9 10..12
{ 1, 4, 7 } { 2, 5, 10 } { 3, 9, 11 } { 6, 8, 12 }
{ 1, 5, 8 } { 2, 6, 11 } { 3, 7, 12 } { 4, 9, 10 }
{ 1, 6, 9 } { 2, 4, 12 } { 3, 8, 10 } { 5, 7, 11 }
----------
```

Notice hows sets which are ranges may be output in range format.

SOCIAL-GOLFERS ☰ [DOWNLOAD]

```
include "partition_set.mzn";
int: weeks;    set of int: WEEK = 1..weeks;
int: groups;   set of int: GROUP = 1..groups;
int: size;     set of int: SIZE = 1..size;
int: ngolfers = groups*size;
set of int: GOLFER = 1..ngolfers;

array[WEEK,GROUP] of var set of GOLFER: Sched;

▶ CONSTRAINTS
▶ SYMMETRY

solve satisfy;

output [ show(Sched[i,j]) ++ " " ++
         if j == groups then "\n" else "" endif |
         i in WEEK, j in GROUP ];
```

Figure 21: Model for the social golfers problems (`social-golfers.mzn`).

3.7 Putting it all together

We finish this section with a complex example illustrating most of the features introduced in this chapter including enumerated types, complex constraints, global constraints, and complex output.

The model of Figure 24 arranges seats at the wedding table. The table has 12 numbered seats in order around the table, 6 on each side. Males must sit in odd numbered seats, and females in even. Ed cannot sit at the end of the table because of a phobia, and the bride and groom must sit next to each other. The aim is to maximize the distance between known hatreds. The distance between seats is the difference in seat number if on the same side, otherwise its the distance to the opposite seat + 1.

Note that in the output statement we consider each seat s and search for a guest g who is assigned to that seat. We make use of the built in function `fix` which checks if a decision variables is fixed and returns its fixed value, and otherwise aborts. This is always safe to use in output statements, since by the time the output statement is run all decision variables should be fixed.

Running

```
$ mzn-g12fd wedding.mzn
```

Results in the output

```
ted bride groom rona bob carol ron alice ed bridesmaid bestman clara
----------
==========
```

CONSTRAINTS ≡

```
constraint
    forall (i in 1..weeks-1) (
              Sched[i,1] < Sched[i+1,1]
    ) /\
    forall (i in WEEK, j in GROUP) (
           card(Sched[i,j]) = size
        /\ forall (k in j+1..groups) (
                Sched[i,j] < Sched[i,k]
             /\ Sched[i,j] intersect Sched[i,k] = {}
           )
    ) /\
    forall (i in WEEK) (
          partition_set([Sched[i,j] | j in GROUP], GOLFER)
       /\ forall (j in 1..groups-1) (
              Sched[i,j] < Sched[i,j+1]
          )
    ) /\
    forall (i in 1..weeks-1, j in i+1..weeks) (
        forall (x,y in GROUP) (
            card(Sched[i,x] intersect Sched[j,y]) <= 1
        )
    );
```

Figure 22: Constraints for the social golfers problems.

SYMMETRY ≡

```
constraint
  % Fix the first week %
  forall (i in GROUP, j in SIZE) (
      ((i-1)*size + j) in Sched[1,i]
  ) /\
  % Fix first group of second week %
  forall (i in SIZE) (
      ((i-1)*size + 1) in Sched[2,1]
  ) /\
  % Fix first 'size' players
  forall (w in 2..weeks, p in SIZE) (
      p in Sched[w,p]
  );
```

Figure 23: Symmetry breaking constraints for the social golfers problems.

WEDDING ≡ [DOWNLOAD]

```
enum Guests = { bride, groom, bestman, bridesmaid, bob, carol,
  ted, alice, ron, rona, ed, clara};
set of int: Seats = 1..12;
set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];
set of Guests: Males = {groom, bestman, bob, ted, ron,ed};
set of Guests: Females = {bride,bridesmaid,carol,alice,rona,clara};

array[Guests] of var Seats: pos; % seat of guest
array[Hatreds] of var Seats: p1;  % seat of guest 1 in hatred
array[Hatreds] of var Seats: p2;  % seat of guest 2 in hatred
array[Hatreds] of var 0..1: sameside;  % seats of hatred on same side
array[Hatreds] of var Seats: cost;  % penalty of hatred

include "alldifferent.mzn";
constraint alldifferent(pos);
constraint forall(g in Males)( pos[g] mod 2 == 1 );
constraint forall(g in Females)( pos[g] mod 2 == 0 );
constraint not (pos[ed] in {1,6,7,12});
constraint abs(pos[bride] - pos[groom]) <= 1 /\
           (pos[bride] <= 6 <-> pos[groom] <= 6);
constraint forall(h in Hatreds)(
               p1[h] = pos[h1[h]] /\
               p2[h] = pos[h2[h]] /\
               sameside[h] = bool2int(p1[h] <= 6 <-> p2[h] <= 6) /\
               cost[h] = sameside[h] * abs(p1[h] - p2[h]) +
                  (1 - sameside[h]) * (abs(13 - p1[h] - p2[h]) + 1) );

solve maximize sum(h in Hatreds)(cost[h]);

output [ show(g)++" " | s in Seats,g in Guests where fix(pos[g]) == s]
       ++ ["\n"];
```

Figure 24: Planning wedding seating using enumerated types (`wedding.mzn`)

The resulting table placement is illustrated in Figure 25 where the lines indicate hatreds. The total distance is 22.

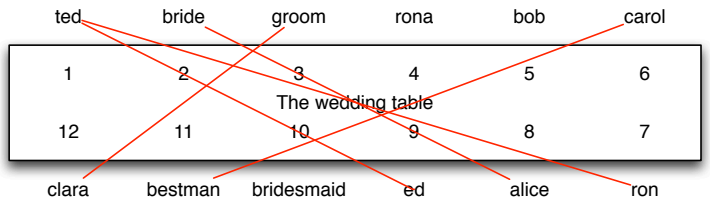


Figure 25: Seating arrangement at the wedding table

Fix

In output items the built-in function `fix` checks that the value of a decision variable is fixed and coerces the instantiation from decision variable to parameter.

4 Predicates and Functions

Predicates in MiniZinc allow us to capture complex constraints of our model in a succinct way. Predicates in MiniZinc are used to model with both predefined global constraints, and to capture and define new complex constraints by the modeller. Functions are used in MiniZinc to capture common structures of models. Indeed a predicate is just a function with output type `var bool`.

4.1 Global Constraints

There are many global constraints defined in MiniZinc for use in modelling. The definitive list is to be found in the documentation for the release, as the list is slowly growing. Below we discuss some of the most important global constraints

4.1.1 Alldifferent

The `alldifferent` constraint takes an array of variables and constrains them to take different values. A use of the `alldifferent` has the form

```
alldifferent(array[int] of var int: x)
```

that is the argument is array of integer variables.

Alldifferent is one of the most studied and used global constraints in constraint programming. It is used to define assignment subproblems, and efficient global propagators for `alldifferent` exist. `send-more-money.mzn` (Figure 11) and `sudoku.mzn` (Figure 12) are examples of models using `alldifferent`.

4.1.2 Cumulative

The `cumulative` constraint is used for describing cumulative resource usage.

```
cumulative(array[int] of var int: s, array[int] of var int: d,
           array[int] of var int: r, var int: b)
```

Requires that a set of tasks given by start times s, durations d, and resource requirements r, never require more than a global resource bound b at any one time.

The model in Figure 26 finds a schedule for moving furniture so that each piece of furniture has enough handlers (people) and enough trolleys available during the move. The available time, handlers and trolleys are given, and the data gives for each object the move duration, the number of handlers and the number of trolleys required. Using the data shown in Figure 27, the command

```
$ mzn-g12fd moving.mzn moving.dzn
```

may result in the output

```
start = [0, 60, 60, 90, 120, 0, 15, 105]
end = 140
----------
==========
```

Figure 28(a) and Figure 28(b) show the requirements for handlers and trolleys at each time in the move for this solution.

MOVING ≡ [DOWNLOAD]

```
include "cumulative.mzn";

enum OBJECTS;
array[OBJECTS] of int: duration; % duration to move
array[OBJECTS] of int: handlers; % number of handlers required
array[OBJECTS] of int: trolleys; % number of trolleys required

int: available_handlers;
int: available_trolleys;
int: available_time;

array[OBJECTS] of var 0..available_time: start;
var 0..available_time: end;

constraint cumulative(start, duration, handlers, available_handlers);
constraint cumulative(start, duration, trolleys, available_trolleys);

constraint forall(o in OBJECTS)(start[o] +duration[o] <= end);

solve minimize end;

output [ "start = \(start)\nend = \(end)\n"];
```

Figure 26: Model for moving furniture using `cumulative` (`moving.mzn`).

MOVING.DZN ≡ [DOWNLOAD]

```
OBJECTS = { piano, fridge, doublebed, singlebed,
            wardrobe, chair1, chair2, table };

duration = [60, 45, 30, 30, 20, 15, 15, 15];
handlers = [3, 2, 2, 1, 2, 1, 1, 2];
trolleys = [2, 1, 2, 2, 2, 0, 0, 1];

available_time = 180;
available_handlers = 4;
available_trolleys = 3;
```

Figure 27: Data for moving furniture using `cumulative` (`moving.dzn`).

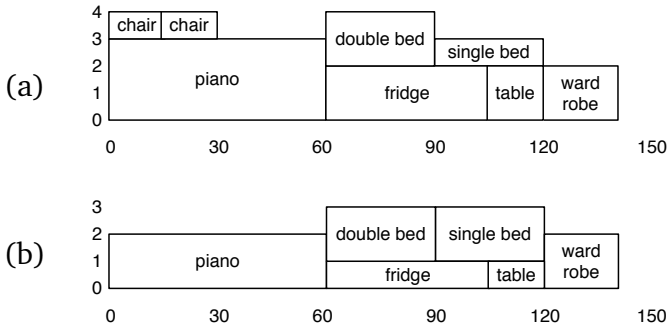


Figure 28: Histograms of usage of (a) handlers and (b) trolleys in the move.

4.1.3 Table

The `table` constraint enforces that the tuple of variables takes a value from a set of tuples. Since there are no tuples in MiniZinc this is encoded using arrays. The usage of `table` has one of the forms

```
table(array[int] of var bool: x, array[int, int] of bool: t)
table(array[int] of var int:  x, array[int, int] of int:  t)
```

depending on whether the tuples are Boolean or integer. The constraint enforces $x \in t$ where we consider x and each row in t to be a tuple, and t to be a set of tuples.

The model in Figure 29 searches for balanced meals. Each meal item has a name (encoded as an integer), a kilojoule count, protein in grams, salt in milligrams, and fat in grams, as well as cost in cents. The relationship between these items is encoded using a `table` constraint. The model searches for a minimal cost meal which has a minimum kilojoule count min_energy, a minimum amount of protein $min\_protein$, maximum amount of salt max_salt and fat max_fat.

4.1.4 Regular

The `regular` constraint is used to enforce that a sequence of variables takes a value defined by a finite automaton. The usage of `regular` has the form

```
regular(array[int] of var int: x, int: Q, int: S,
        array[int,int] of int: d, int: q0, set of int: F)
```

It constrains that the sequence of values in array x (which must all be in the range $1..S$) is accepted by the DFA of Q states with input $1..S$ and transition function d (which maps $< 1..Q, 1..S >$ to $0..Q$) and initial state $q0$ (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). State 0 is reserved to be an always failing state.

Consider a nurse rostering problem. Each nurse is scheduled for each day as either: (d) on day shift, (n) on night shift, or (o) off. In each four day period a nurse must have at least one day off, and no nurse can be scheduled for 3 night shifts in a row. This can be encoded using the incomplete DFA shown in Figure 31. We can encode this DFA as having start state 1, final states 1 .. 6, and transition

MEAL ☰ [DOWNLOAD]

```
% Planning a balanced meal
include "table.mzn";
int: min_energy;
int: min_protein;
int: max_salt;
int: max_fat;
set of FOOD: desserts;
set of FOOD: mains;
set of FOOD: sides;
enum FEATURE = { name, energy, protein, salt, fat, cost};
enum FOOD;
array[FOOD,FEATURE] of int: dd; % food database

array[FEATURE] of var int: main;
array[FEATURE] of var int: side;
array[FEATURE] of var int: dessert;
var int: budget;

constraint main[name] in mains;
constraint side[name] in sides;
constraint dessert[name] in desserts;
constraint table(main, dd);
constraint table(side, dd);
constraint table(dessert, dd);
constraint main[energy] + side[energy] + dessert[energy] >=min_energy;
constraint main[protein]+side[protein]+dessert[protein] >=min_protein;
constraint main[salt] + side[salt] + dessert[salt] <= max_salt;
constraint main[fat] + side[fat] + dessert[fat] <= max_fat;
constraint budget = main[cost] + side[cost] + dessert[cost];

solve minimize budget;

output ["main = ",show(to_enum(FOOD,main[name])),",
        ", side = ",show(to_enum(FOOD,side[name])),
        ", dessert = ",show(to_enum(FOOD,dessert[name]))],
        ", cost = ",show(budget), "\n"];
```

Figure 29: Model for meal planning using `table` constraint (`meal.mzn`).

MEAL.DZN ≡ [DOWNLOAD]

```
FOODS = { icecream, banana, chocolatecake, lasagna,
          steak, rice, chips, brocolli, beans} ;

dd = [| icecream,      1200,  50,  10, 120,  400     % icecream
      | banana,         800, 120,   5,  20,  120     % banana
      | chocolatecake, 2500, 400,  20, 100,  600     % chocolate cake
      | lasagna,       3000, 200, 100, 250,  450     % lasagna
      | steak,         1800, 800,  50, 100, 1200     % steak
      | rice,          1200,  50,   5,  20,  100     % rice
      | chips,         2000,  50, 200, 200,  250     % chips
      | brocolli,       700, 100,  10,  10,  125     % brocolli
      | beans,         1900, 250,  60,  90,  150 |]; % beans

min_energy = 3300;
min_protein = 500;
max_salt = 180;
max_fat = 320;
desserts = { icecream, banana, chocolotecake };
mains = { lasagna, steak, rice };
sides = { chips, brocolli, beans };
```

Figure 30: Data for meal planning defining the `table` used (`meal.dzn`).

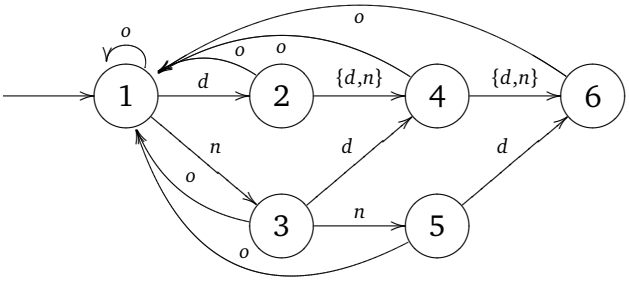


Figure 31: A DFA determining correct rosters.

function

	d	n	o
1	2	3	1
2	4	4	1
3	4	5	1
4	6	6	1
5	6	0	1
6	0	0	1

Note that state 0 in the table indicates an error state. The model shown in Figure 32 finds a schedule for *num_nurses* nurses over *num_days* days, where we require *req_day* nurses on day shift each day, and *req_night* nurses on night shift, and that each nurse takes at least *min_night* night shifts.

Running the command

```
$ mzn-g12fd nurse.mzn nurse.dzn
```

finds a 10 day schedule for 7 nurses, requiring 3 on each day shift and 2 on each night shift, with a minimum 2 night shifts per nurse. A possible output is

```
o d n n o n n d o o
d o n d o d n n o n
o d d o d o d n n o
d d d o n n d o n n
d o d n n o d o d d
n n o d d d o d d d
n n o d d d o d d d
----------
```

There is an alternate form of the regular constraint `regular_nfa` which specifies the regular expression using and NFA (without ϵ arcs). This constraint has the form

```
regular_nfa(array[int] of var int: x, int: Q, int: S,
        array[int,int] of set of int: d, int: q0, set of int: F)
```

It constrains that the sequence of values in array x (which must all be in the range $1..S$) is accepted by the NFA of Q states with input $1..S$ and transition function d (which maps $< 1..Q, 1..S >$ to subsets of $1..Q$) and initial state $q0$ (which must be in $1..Q$) and accepting states F (which all must be in $1..Q$). There is no need for a failing state 0, since the transition function can map to an empty set of states.

4.2 Defining Predicates

One of the most powerful modelling features of MiniZinc is the ability for the modeller to define their own high-level constraints. This allows them to abstract and modularise their model. It also allows re-use of constraints in different models and allows the development of application specific libraries defining the standard constraints and types.

We start with a simple example, revisiting the job shop scheduling problem from the previous section. The model is shown in Figure 33. The item of interest is the `predicate` item:

NOOVERLAP ≡

```
predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;
```

This defines a new constraint that enforces that a task with start time `s1` and duration `d1` does not overlap with a task with start time `s2` and duration `d2`. This can now be used inside the model anywhere any other Boolean expression (involving decision variables) can be used.

As well as predicates the modeller can define new constraints that only involve parameters. Unlike predicates these can be used inside the test of a conditional expression. These are defined using the keyword `test`. For example

```
test even(int:x) = x mod 2 = 0;
```

NURSE ≡ [DOWNLOAD]

```
% Simple nurse rostering
include "regular.mzn";
enum NURSE;
enum DAY;
int: req_day;
int: req_night;
int: min_night;

enum SHIFT = { d, n, o };
int: S = card(SHIFT);

int: Q = 6; int: q0 = 1; set of int: STATE = 1..Q;
array[STATE,SHIFT] of int: t =
     [| 2, 3, 1    % state 1
      | 4, 4, 1    % state 2
      | 4, 5, 1    % state 3
      | 6, 6, 1    % state 4
      | 6, 0, 1    % state 5
      | 0, 0, 1|]; % state 6

array[NURSE,DAY] of var SHIFT: roster;

constraint forall(j in DAY)(
             sum(i in NURSE)(roster[i,j] == d) == req_day /\
             sum(i in NURSE)(roster[i,j] == n) == req_night
           );
constraint forall(i in NURSE)(
             regular([roster[i,j] | j in DAY], Q, S, t, q0, STATE) /\
             sum(j in DAY)(roster[i,j] == n) >= min_night
           );

solve satisfy;

output [ show(roster[i,j]) ++ if j==card(DAY) then "\n" else " " endif
       | i in NURSE, j in DAY ];
```

Figure 32: Model for nurse rostering using `regular` constraint (`nurse.mzn`).

JOBSHOP2 ≡ [DOWNLOAD]

```
int: jobs;                                      % no of jobs
set of int: JOB = 1..jobs;
int: tasks;                                     % no of tasks per job
set of int: TASK = 1..tasks;
array [JOB,TASK] of int: d;                     % task durations
int: total = sum(i in JOB, j in TASK)(d[i,j]);% total duration
int: digs = ceil(log(10.0,total));              % digits for output
array [JOB,TASK] of var 0..total: s;            % start times
var 0..total: end;                              % total end time

► NOOVERLAP

constraint %% ensure the tasks occur in sequence
    forall(i in JOB) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );

constraint %% ensure no overlap of tasks
    forall(j in TASK) (
        forall(i,k in JOB where i < k) (
            no_overlap(s[i,j], d[i,j], s[k,j], d[k,j])
        )
    );

solve minimize end;

output ["end = \(end)\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in JOB, j in TASK ];
```

Figure 33: Model for job shop scheduling using predicates (`jobshop2.mzn`).

Predicate definitions

Predicates are defined by a statement of the form

predicate ⟨pred-name⟩ (⟨arg-def⟩, ..., ⟨arg-def⟩) = ⟨bool-exp⟩

The ⟨pred-name⟩ must be a valid MiniZinc identifier, and each ⟨arg-def⟩ is a valid MiniZinc type declaration.
One relaxation of argument definitions is that the index types for arrays can be unbounded written `int`.
Similarly, tests are defined by a statement of the form

test ⟨pred-name⟩ (⟨arg-def⟩, ..., ⟨arg-def⟩) = ⟨bool-exp⟩

The ⟨bool-exp⟩ of the body must be fixed.
We also introduce a new form of the `assert` command for use in predicates.

assert (⟨bool-exp⟩, ⟨string-exp⟩, ⟨exp⟩)

The type of the `assert` expression is the same as the type of the last argument. The `assert` expression checks whether the first argument is false, and if so prints the second argument string. If the first argument is true it returns the third argument.

Note that assert expressions are lazy in the third argument, that is if the first argument is false they will not be evaluated. Hence, they can be used for checking

```
predicate lookup(array[int] of var int:x, int: i, var int: y) =
    assert(i in index_set(x), "index out of range in lookup"
           y = x[i]
    );
```

This code will not evaluate $x[i]$ if i is out of the range of the array x.

4.3 Defining Functions

Functions are defined in MiniZinc similarly to predicates, but with a more general return type.

The function below defines the row in a Sudoku matrix of the $a1^{th}$ row of the a^{th} of subsquares

```
function int: posn(int: a, int: a1) = (a-1) * S + a1;
```

With this definition we can replace the last constraint in the Sudoku problem shown in Figure 12 by

```
constraint forall(a, o in SubSquareRange)(
                  alldifferent([ puzzle [ posn(a,a0), posn(o,o1) ] |
                                          a1,o1 in SubSquareRange ] ) );
```

Functions are useful for encoding complex expressions that are used frequently in the model. For example, imagine placing the numbers 1 to n in different positions in an $n \times n$ grid such that the Manhattan distance between any two numbers i and j is greater than the maximum of the two numbers minus 1. The aim is to minimize the total of the Manhattan distances between the pairs. The Manhattan distance function can be expressed as:

MANHATTAN ≡ [DOWNLOAD]

```
int: n;
set of int: NUM = 1..n;

array[NUM] of var NUM: x;
array[NUM] of var NUM: y;
array[NUM,NUM] of var 0..2*n-2: dist =
     array2d(NUM,NUM,[
     if i < j then manhattan(x[i],y[i],x[j],y[j]) else 0 endif
                     | i,j in NUM ]);

► MANF

constraint forall(i,j in NUM where i < j)
                 (dist[i,j] >= max(i,j)-1);

var int: obj = sum(i,j in NUM where i < j)(dist[i,j]);
solve minimize obj;

% simply to display result
include "alldifferent_except_0.mzn";
array[NUM,NUM] of var 0..n: grid;
constraint forall(i in NUM)(grid[x[i],y[i]] = i);
constraint alldifferent_except_0([grid[i,j] | i,j in NUM]);

output ["obj = \(obj);\n"] ++
       [ if fix(grid[i,j]) > 0 then show(grid[i,j]) else "." endif
         ++ if j = n  then "\n" else "" endif
       | i,j in NUM ];
```

Figure 34: Model for a number placement problem illustrating the use of functions (`manhattan.mzn`).

MANF ≡

```
function var int: manhattan(var int: x1, var int: y1,
                            var int: x2, var int: y2) =
         abs(x1 - x2) + abs(y1 - y2);
```

The complete model is shown in Figure 34.

Function definitions

Functions are defined by a statement of the form

function ⟨ret-type⟩ : ⟨func-name⟩ (⟨arg-def⟩, ..., ⟨arg-def⟩) = ⟨exp⟩

The ⟨func-name⟩ must be a valid MiniZinc identifier, and each ⟨arg-def⟩ is a valid MiniZinc type declaration. The ⟨ret-type⟩ is the return type of the function which must be the type of ⟨exp⟩. Arguments have the same restrictions as in predicate definitions.

Functions in MiniZinc can have any return type, not just fixed return types. Functions are useful for defining and documenting complex expressions that are used mulitple times in a model.

4.4 Reflection Functions

To help write generic tests and predicates, various reflection functions return information about array index sets, var set domains and decision variable ranges. Those for index sets are `index_set(`⟨*1-D array*⟩`)`, `index_set_1of2(`⟨*2-D array*⟩`)`, `index_set_2of2(`⟨*2-D array*⟩`)` and so on for higher dimensional arrays.

A better model of the job shop conjoins all the non-overlap constraints for a single machine into a single disjunctive constraint. An advantage of this approach is that while we may initially model this simply as a conjunction of non-overlap, if the underlying solver has a better approach to solving disjunctive constraints we can use that instead, with minimal changes to our model. The model is shown in Figure 35.

The `disjunctive` constraint takes an array of start times for each task and an array of their durations and makes sure that only one task is active at any one time. We define the disjunctive constraint as a predicate with signature

```
predicate disjunctive(array[int] of var int:s, array[int] of int:d);
```

We can use the disjunctive constraint to define the non-overlap of tasks as shown in Figure 35. We assume a definition for the `disjunctive` predicate is given by the file `disjunctive.mzn` which is included in the model. If the underlying system supports `disjunctive` directly, it will include a file `disjunctive.mzn` in its globals directory (with contents just the signature definition above). If the system we are using does not support disjunctive directly we can give our own definition by creating the file `disjunctive.mzn`. The simplest implementation simply makes use of the `no_overlap` predicate defined above. A better implementation is to make use of a global `cumulative` constraint assuming it is supported by the underlying solver. Figure 36 shows an implementation of `disjunctive`. Note how we use the `index_set` reflection function to (a) check that the arguments to `disjunctive` make sense, and (b) construct the array of resource utilisations of the appropriate size for `cumulative`. Note also that we use a ternary version of `assert` here

4.5 Local Variables

It is often useful to introduce *local variables* in a predicate, function or test. The `let` expression allows you to do so. It can be used to introduce both decision variables and parameters, but parameters must be initialised. For example:

```
var s..e: x;
let {int: l = s div 2; int: u = e div 2; var l .. u: y;} in x = 2*y
```

JOBSHOP3 ≡ [DOWNLOAD]

```
include "disjunctive.mzn";

int: jobs;                                  % no of jobs
set of int: JOB = 1..jobs;
int: tasks;                                 % no of tasks per job
set of int: TASK = 1..tasks;
array [JOB,TASK] of int: d;                 % task durations
int: total = sum(i in JOB, j in TASK)(d[i,j]);% total duration
int: digs = ceil(log(10.0,total));          % digits for output
array [JOB,TASK] of var 0..total: s;        % start times
var 0..total: end;                          % total end time

constraint %% ensure the tasks occur in sequence
    forall(i in JOB) (
        forall(j in 1..tasks-1)
            (s[i,j] + d[i,j] <= s[i,j+1]) /\
        s[i,tasks] + d[i,tasks] <= end
    );

constraint %% ensure no overlap of tasks
    forall(j in TASK) (
        disjunctive([s[i,j] | i in JOB], [d[i,j] | i in JOB])
    );

solve minimize end;

output ["end = \(end)\n"] ++
       [ show_int(digs,s[i,j]) ++ " " ++
         if j == tasks then "\n" else "" endif |
         i in JOB, j in TASK ];
```

Figure 35: Model for job shop scheduling using `disjunctive` predicate (`jobshop3.mzn`).

introduces parameters l and u and variable y. While most useful in predicate, function and test definitions, let expressions can also be used in other expressions, for example for eliminating common subexpressions:

```
constraint let { var int: s = x1 + x2 + x3 + x4 } in
           l <= s /\ s <= u;
```

Local variables can be used anywhere and can be quite useful, for simplifying complex expressions. Figure 37 gives a revised version of the wedding model, using local variables to define the objective

DISJUNCTIVE ≡ [DOWNLOAD]

```
include "cumulative.mzn";

predicate disjunctive(array[int] of var int:s, array[int] of int:d) =
          assert(index_set(s) == index_set(d), "disjunctive: " ++
          "first and second arguments must have the same index set",
            cumulative(s, d, [ 1 | i in index_set(s) ], 1)
          );
```

Figure 36: Defining a `disjunctive` predicate using `cumulative` (`disjunctive.mzn`).

function, rather than adding lots of variables to the model explicitly.

4.6 Context

One limitation is that predicates and functions containing decision variables that are not initialised in the declaration cannot be used inside a negative context. The following is illegal

```
predicate even(var int:x) =
          let { var int: y } in x = 2 * y;

constraint not even(z);
```

The reason for this is that solvers only solve existentially constrained problems, and if we introduce a local variable in a negative context, then the variable is *universally quantified* and hence out of scope of the underlying solvers. For example the $\neg even(z)$ is equivalent to $\neg\exists y.z = 2y$ which is equivalent to $\forall y.z \neq 2y$.

If local variables are given values, then they can be used in negative contexts. The following is legal

```
predicate even(var int:x) =
          let { var int: y = x div 2; } in x = 2 * y;

constraint not even(z);
```

Note that the meaning of even is correct, since if x is even then $x = 2 * (x \text{ div } 2)$. Note that for this definition $\neg even(z)$ is equivalent to $\neg\exists y.y = z \; div \; 2 \wedge z = 2y$ which is equivalent to $\exists y.y = z \; div \; 2 \wedge \neg z \neq 2y$, because y is functionally defined by z.

Every expression in MiniZinc appears in one of the four *contexts*: root, positive, negative, or mixed. The context of a non-Boolean expression is simply the context of its nearest enclosing Boolean expression. The one exception is that the objective expression appears in a root context (since it has no enclosing Boolean expression).

For the purposes of defining contexts we assume implication expressions e `->` e' are rewritten equivalently as `not` e `\/` e', and similarly e `<-` e' are rewritten as e `->` `not` e'.

The context for a Boolean expression is given by: MiniZinc:

WEDDING2 ≡ [DOWNLOAD]

```
enum Guests = { bride, groom, bestman, bridesmaid, bob, carol,
  ted, alice, ron, rona, ed, clara};
set of int: Seats = 1..12;
set of int: Hatreds = 1..5;
array[Hatreds] of Guests: h1 = [groom, carol, ed, bride, ted];
array[Hatreds] of Guests: h2 = [clara, bestman, ted, alice, ron];
set of Guests: Males = {groom, bestman, bob, ted, ron,ed};
set of Guests: Females = {bride,bridesmaid,carol,alice,rona,clara};

array[Guests] of var Seats: pos; % seat of guest

include "alldifferent.mzn";
constraint alldifferent(pos);

constraint forall(g in Males)( pos[g] mod 2 == 1 );
constraint forall(g in Females)( pos[g] mod 2 == 0 );

constraint not (pos[ed] in {1,6,7,12});
constraint abs(pos[bride] - pos[groom]) <= 1 /\
           (pos[bride] <= 6 <-> pos[groom] <= 6);

solve maximize sum(h in Hatreds)(
          let {  var Seats: p1 = pos[h1[h]];
                 var Seats: p2 = pos[h2[h]];
                 var 0..1: same = bool2int(p1 <= 6 <-> p2 <= 6); } in
          same * abs(p1 - p2) + (1-same) * (abs(13 - p1 - p2) + 1));

output [ show(g)++" " | s in Seats,g in Guests where fix(pos[g]) == s]
       ++ ["\n"];
```

Figure 37: Using local variables to define a complex objective function (`wedding2.mzn`)

root root context is the context for any expression e appearing as the argument of `constraint` or as an assignment item, or appearing as an sub expression e or e' in an expression e `/\` e' occuring in a root context.

Root context Boolean expressions must hold in any model of the problem.

positive positive context is the context for any expression appearing as a sub expression e or e' in an expression e `\/` e' occuring in a root or positive context, appearing as a sub expression e or e' in an expression e `/\` e' occuring in a positive context, or appearing as a sub expression e in an expression `not` e appearing in a negative context.

Positive context Boolean expressions need not hold in a model, but making them hold will only increase the possibility that the enclosing constraint holds. A positive context expression has an even number of negations in the path from the enclosing root context to the expression.

negative positive context is the context for any expression appearing in a root context, or appearing as a sub expression e or e' in an expression e `\/` e' or e `/\` e' occuring in a negative, or appearing as a sub expression e in an expression `not` e appearing in a positive context.

Negative context Boolean expressions need not hold in a model, but making them false will increase the possibility that the enclosing constraint holds. A negative context expression has an odd number of negations in the path from the enclosing root context to the expression.

mixed mixed context is the context for any Boolean expression appearing as a subexpression e or e' in e `<->` e', e `=` e', or `bool2int`(e),

Mixed context expression are effectively both positive and negative. This can be seen from the fact that $e \leftrightarrow e'$ is equivalent to $(e \wedge e') \vee (\neg e \vee \neq e')$ and `bool2int`(e) is equivalent to $(e \wedge x = 1) \vee (\neg e \wedge x = 0)$.

Consider the code fragment

```
constraint x > 0 /\ (i <= 4 -> x + bool2int(x > i) = 5);
```

then `x > 0` is in the root context, `i <= 4` is in a negative context, `x + bool2int(x > i) = 5` is in a positive context, and `x > i` is in a mixed context.

4.7 Local Constraints

Let expressions can also be used to include local constraints, usually to constrain the behaviour of local variables. For example, consider defining a square root function making use of only multiplication:

```
function var float: mysqrt(var float:x) =
         let { var float: y;
               constraint y >= 0;
               constraint x = y * y; } in y;
```

The local constraints ensure `y` takes the correct value; which is then returned by the function.

Local constraint can be used in any let expression, though the most common usage is in defining functions.

Let expressions

Local variables can be introduced in any expression with a *let expression* of the form:

`let` { ⟨dec⟩; ... ⟨dec⟩; } **`in`** ⟨*exp*⟩

The declarations ⟨dec⟩ can be declarations of decision variables and parameters (which must be initialised) or constraint items. No declaration can make use of a newly declared variable before it is introduced.

Note that local variables and constraints cannot occur in tests. Local variables cannot occur in predicates or functions that appear in a negative or mixed context, unless the variable is defined by an expression.

REFLECTION ≡ [DOWNLOAD]

```
var -10..10: x;
constraint x in 0..4;
int: y = lb(x);
set of int: D = dom(x);
solve satisfy;
output ["y = ", show(y), "\nD = ", show(D), "\n"];
```

Figure 38: Using reflection predicates (`reflection.mzn`)

4.8 Domain Reflection Functions

Other important reflection functions are those that allow us to access the domains of variables. The expression `lb(x)` returns a value that is lower than or equal to any value that x may take in a solution of the problem. Usually it will just be the declared lower bound of x. If x is declared as a non-finite type, e.g. simply `var int` then it is an error. Similarly the expression `dom(x)` returns a (non-strict) superset of the possible values of x in any solution of the problem. Again it is usually the declared values, and again if it is not declared as finite then there is an error.

For example, the model show in Figure 38 may output

```
y = -10
D = -10..10
----------
```

or

```
y = 0
D = {0, 1, 2, 3, 4}
----------
```

or any answer with $-10 \leq y \leq 0$ and $\{0,\ldots,4\} \subseteq D \subseteq \{-10,\ldots,10\}$.

Variable domain reflection expressions should be used in a manner where they are correct for any safe approximations, but note this is not checked! For example the additional code

```
var -10..10: z;
constraint z <= y;
```

is not a safe usage of the domain information. Since using the tighter (correct) approximation leads to more solutions than the weaker initial approximation.

Domain reflection

There are reflection functions to interrogate the possible values of expressions containing variables:

- `dom ( ⟨exp⟩ )`: returns a safe approximation to the possible values of the expression.
- `lb ( ⟨exp⟩ )`: returns a safe approximation to the lower bound value of the expression.
- `ub ( ⟨exp⟩ )`: returns a safe approximation to the upper bound value of the expression.

The expressions for `lb` and `ub` can only be of types `int`, `bool`, `float` or `set of int`. For `dom` the type cannot be `float`. If one of the variables appearing in *exp* has a non-finite declared type (e.g. `var int` or `var float` type) then an error can occur.
There are also versions that work directly on arrays of expressions (with similar restrictions):

- `dom_array ( ⟨array-exp⟩ )`: returns a safe approximation to the union of all possible values of the expressions appearing in the array.
- `lb_array ( ⟨array-exp⟩ )`: returns a safe approximation to the lower bound of all expressions appearing in the array.
- `ub_array ( ⟨array-exp⟩ )`: returns a safe approximation to the upper bound of all expressions appearing in the array.

The combinations of predicates, local variables and domain reflection allows the definition of complex global constraints by decomposition. We can define the time based decomposition of the `cumulative` constraint using the code shown in Figure 39.

The decomposition uses `lb` and `ub` to determine the set of times `times` over which tasks could range. It then asserts for each time t in `times` that the sum of resources for the active tasks at time t is less than the bound b.

4.9 Scope

It is worth briefly mentioning the scope of declarations in MiniZinc. MiniZinc has a single namespace, so all variables appearing in declarations are visible in every expression in the model. MiniZinc introduces locally scoped variables in a number of ways:

- as iterator variables in comprehension expressions
- using `let` expressions
- as predicate and function arguments

Any local scoped variable overshadows the outer scoped variables of the same name.

For example, in the model shown in Figure 40 the `x` in `-x <= y` is the global `x`, the `x` in `even(x)` is the iterator `x in 1..u`, while the `y` in the disjunction is the second argument of the predicate.

CUMULATIVE ≡ [DOWNLOAD]

```
%-----------------------------------------------------------------------%
% Requires that a set of tasks given by start times 's',
% durations 'd', and resource requirements 'r',
% never require more than a global
% resource bound 'b' at any one time.
% Assumptions:
% - forall i, d[i] >= 0 and r[i] >= 0
%-----------------------------------------------------------------------%
predicate cumulative(array[int] of var int: s,
                     array[int] of var int: d,
                     array[int] of var int: r, var int: b) =
   assert(index_set(s) == index_set(d) /\
          index_set(s) == index_set(r),
     "cumulative: the array arguments must have identical index sets",
   assert(lb_array(d) >= 0 /\ lb_array(r) >= 0,
     "cumulative: durations and resource usages must be non-negative",
          let {
              set of int: times =
                lb_array(s) ..
                max([ ub(s[i]) + ub(d[i]) | i in index_set(s) ])
              }
          in
             forall( t in times ) (
                b >= sum( i in index_set(s) ) (
                   bool2int( s[i] <= t /\ t < s[i] + d[i] ) * r[i]
                )
             )
       )
   );
```

Figure 39: Defining a `cumulative` predicate by decomposition (`cumulative.mzn`).

5 Option Types

Option types are a powerful abstraction that allow concise modelling. An option type decision variable represents a decision that has another possibility $\top$, represented in MiniZinc as `<>` indicating the variable is *absent*. Option type decisions are useful for modelling problems where a decision is not meaningful unless other decisions are made first.

SCOPE ☰ [DOWNLOAD]

```
int: x = 3;
int: y = 4;
predicate smallx(var int:y) = -x <= y /\ y <= x;
predicate p(int: u, var bool: y) =
      exists(x in 1..u)(y \/ smallx(x));
constraint p(x,false);
solve satisfy;
```

Figure 40: A model for illustrating scopes of variables (`scope.mzn`)

5.1 Declaring and Using Option Types

Option type Variables

An option type variable is declared as:

[**var**] **opt** ⟨*type*⟩ : ⟨*var-name*⟩

where *type* is one of `int`, `float` or `bool` or a fixed range expression. Option type variables can be parameters but this is rarely useful.

An option type variable can take the addition value <> indicating *absent*

Three builtin functions are provided for option type variables: `absent`(v) returns `true` iff option type variable v takes the value <>, `occurs`(v) returns `true` iff option type variable v does not take the value <>, and `deopt`(v) returns the normal value of v or fails if it takes the value <>.

The most common use of option types is for optional tasks in scheduling. In the flexible job shop scheduling problem we have n tasks to perform on k machines, and the time to complete each task on each machine may be different. The aim is to minimize the completion time of all tasks. A model using option types to encode the problem is given in Figure 41. We model the problem using $n \times k$ optional tasks representing the possibility of each task run on each machine. We require that start time of the task and its duration spans the optional tasks that make it up, and require only one actually runs using the `alternative` global constraint. We require that at most one task runs on any machine using the `disjunctive` global constraint extended to optional tasks. Finally we constrain that at most k tasks run at any time, a redundant constraint that holds on the actual (not optional) tasks.

5.2 Hidden Option Types

Option type variable arise implicitly when list comprehensions are constructed with iteration over variable sets, or where the expressions in `where` clauses are not fixed.

For example the model fragment

```
var set of 1..n: x;
constraint sum(i in x)(i) <= limit;
```

is syntactic sugar for

```
var set of 1..n: x;
constraint sum(i in 1..n)(if i in x then i else <> endif) <= limit;
```

FLEXIBLE-JS ≡ [DOWNLOAD]

```
int: horizon;                  % time horizon
set of int: Time = 0..horizon;
enum Task;
enum Machine;

array[Task,Machine] of int: d; % duration on each machine
int: maxd = max([ d[t,m] | t in Task, m in Machine ]);
int: mind = min([ d[t,m] | t in Task, m in Machine ]);

array[Task] of var Time: S;             % start time
array[Task] of var mind..maxd: D;       % duration
array[Task,Machine] of var opt Time: O; % optional task start

constraint forall(t in Task)(alternative(S[t],D[t],
                [O[t,m]|m in Machine],[d[t,m]|m in Machine]));
constraint forall(m in Machine)
                (disjunctive([O[t,m]|t in Task],[d[t,m]|t in Task]));
constraint cumulative(S,D,[1|i in Task],k);

solve minimize max(t in Task)(S[t] + D[t]);
```

Figure 41: Model for flexible job shop scheduling using option types (`flexible-js.mzn`).

The `sum` builtin function actually operates on a list of type-inst `var opt int`. Since the <> acts as the identity 0 for + this gives the expected results.

Similarly the model fragment

```
array[1..n] of var int: x;
constraint forall(i in 1..n where x[i] >= 0)(x[i] <= limit);
```

is syntactic sugar for

```
array[1..n] of var int: x;
constraint forall(i in 1..n)(if x[i] >= 0 then x[i] <= limit else <> endif);
```

Again the `forall` function actually operates on a list of type-inst `var opt bool`. Since <> acts as identity `true` for ∧ this gives the expected results.

The hidden uses can lead to unexpected behaviour though so care is warranted. Consider

```
var set of 1..9: x;
constraint card(x) <= 4;
constraint length([ i | i in x]) > 5;
solve satisfy;
```

which would appear to be unsatisfiable. It returns $x = 1, 2, 3, 4$ as example answer. This is correct since the second constraint is equivalent to

```
constraint length([ if i in x then i else <> endif | i in 1..9 ]) > 5;
```

and the length of the list of optional integers is always 9 so the constraint always holds!

One can avoid hidden option types by not constructing iteration over variables sets or using unfixed `where` clauses. For example the above two examples could be rewritten without option types as

```
var set of 1..n: x;
constraint sum(i in 1..n)(bool2int(i in x)*i) <= limit;
```

and

```
array[1..n] of var int: x;
constraint forall(i in 1..n)(x[i] >= 0 -> x[i] <= limit);
```

6 Search

By default in MiniZinc there is no declaration of how we want to search for solutions. This leaves the search completely up to the underlying solver. But sometimes, particularly for combinatorial integer problems, we may want to specify how the search should be undertaken. This requires us to communicate to the solver a search strategy. Note that the search strategy is *not* really part of the model. Indeed it is not required that each solver implements all possible search strategies. MiniZinc uses a consistent approach to communicating extra information to the constraint solver using *annotations*.

6.1 Finite Domain Search

Search in a finite domain solver involves examining the remaining possible values of variables and choosing to constrain some variables further. The search then adds a new constraint that restricts the remaining values of the variable (in effect guessing where the solution might lie), and then applies propagation to determine what other values are still possible in solutions. In order to guarantee completeness, the search leaves another choice which is the negation of the new constraint. The search ends either when the finite domain solver detects that all constraints are satisfied, and hence a solution has been found, or that the constraints are unsatisfiable. When unsatisfiability is detected the search must proceed down a different set of choices. Typically finite domain solvers use *depth first search* where they undo the last choice made and then try to make a new choice.

NQUEENS ≡ [DOWNLOAD]

```
int: n;
array [1..n] of var 1..n: q; % queen is column i is in row q[i]

include "alldifferent.mzn";

constraint alldifferent(q);                        % distinct rows
constraint alldifferent([ q[i] + i | i in 1..n]); % distinct diagonals
constraint alldifferent([ q[i] - i | i in 1..n]); % upwards+downwards

► SEARCH
output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

Figure 42: Model for n-queens (`nqueens.mzn`).

A simple example of a finite domain problem is the n queens problem which requires that we place n queens on an $n \times n$ chessboard so that none can attack another. The variable $q[i]$ records in which row the queen in column i is placed. The `alldifferent` constraints ensure that no two queens are on the same row, or diagonal. A typical (partial) search tree for $n = 9$ is illustrated in the left of Figure 43. We first set $q[1] = 1$, this removes values from the domains of other variables, so that e.g. $q[2]$ cannot take the values 1 or 2. We then set $q[2] = 3$, this further removes values from the domains of other variables. We set $q[3] = 5$ (its earliest possible value). The state of the chess board

after these three decisions is shown in Figure 43(a) where the queens indicate the position of the queens fixed already and the stars indicate positions where we cannot place a queen since it would be able to take an already placed queen.

A search strategy determines which choices to make. The decisions we have made so far follow the simple strategy of picking the first variable which is not fixed yet, and try to set it to its least possible value. Following this strategy the next decision would be $q[4] = 7$. An alternate strategy for variable selection is to choose the variable whose current set of possible values (*domain*) is smallest. Under this so called *first-fail* variable selection strategy the next decision would be $q[6] = 4$. If we make this decision, then initially propagation removes the additional values shown in Figure 43(b). But this leaves only one value for $q[8]$, $q[8] = 7$, so this is forced, but then this leaves only one possible value for $q[7]$ and $q[9]$, that is 2. Hence a constraint must be violated. We have detected unsatisfiability, and the solver must backtrack undoing the last decision $q[6] = 4$ and adding its negation $q[6] \neq 4$ (leading us to state (c) in the tree in Figure 43) which forces $q[6] = 8$. This removes some values from the domain and then we again reinvoke the search strategy to decide what to do.

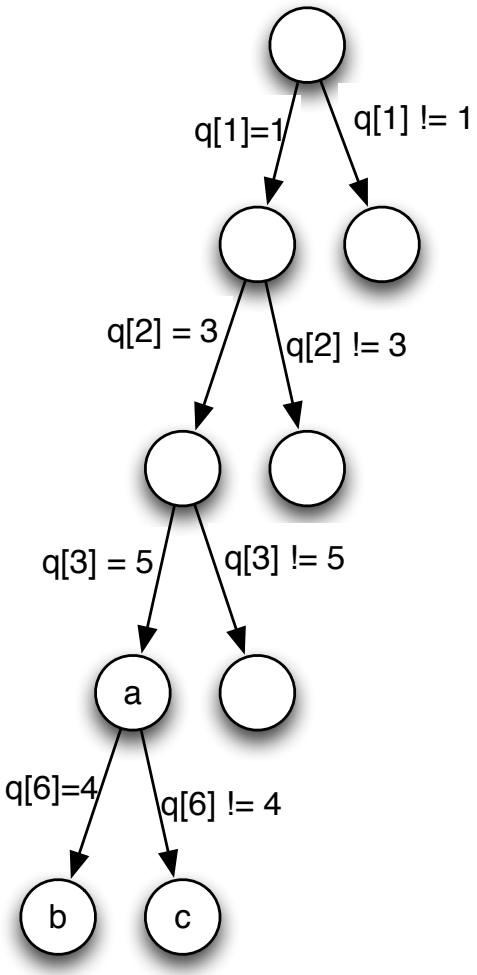
Many finite domain searches are defined in this way: choose a variable to constrain further, and then choose how to constrain it further.

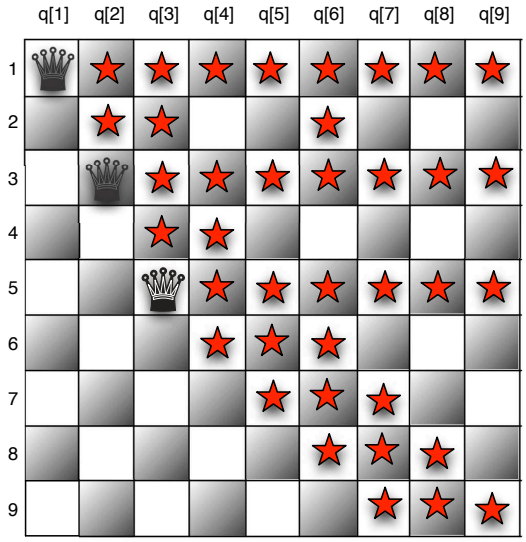
6.2 Search Annotations

Search annotations in MiniZinc specify how to search in order to find a solution to the problem. The annotation is attached to the solve item, after the keyword `solve`. The search annotation

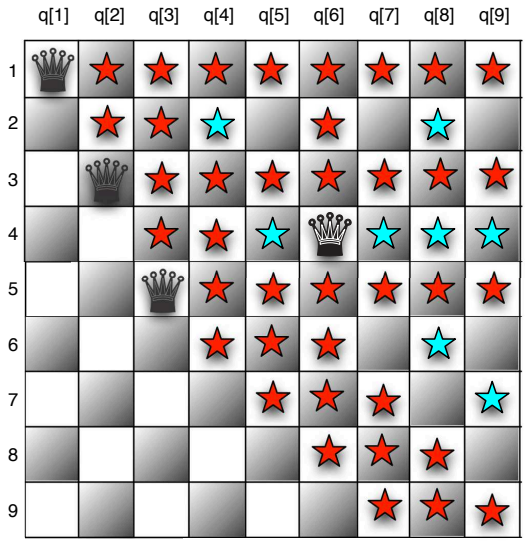
```
SEARCH ≡
  solve :: int_search(q, first_fail, indomain_min, complete)
        satisfy;
```

appears on the solve item. Annotations are attached to parts of the model using the connector `::`. This search annotation means that we should search by selecting from the array of integer variables `q`, the variable with the smallest current domain (this is the `first_fail` rule), and try setting it to its smallest possible value (`indomain_min` value selection), looking across the entire search tree (`complete` search).





(a)



(b)

Figure 43: Partial search trees for 9 queens: (a) the state after the addition of $q[1] = 1$, $q[2] = 4$, $q[3] = 5$ (b) the initial propagation on adding further $q[6] = 4$.

Basic search annotations

There are three basic search annotations corresponding to different basic variable types:

- `int_search(`*`variables`*`, `*`varchoice`*`, `*`constrainchoice`*`, `*`strategy`*`)` where *variables* is an one dimensional array of `var int`, *varchoice* is a variable choice annotation discusses below, *constrainchoice* is a choice of how to constrain a variable, discussed below, and *strategy* is a search strategy which we will assume for now is `complete`.
- `bool_search(`*`variables`*`, `*`varchoice`*`, `*`constrainchoice`*`, `*`strategy`*`)` where *variables* is an one dimensional array of `var bool` and the rest are as above.
- `set_search(`*`variables`*`, `*`varchoice`*`, `*`constrainchoice`*`, `*`strategy`*`)` where *variables* is an one dimensional array of `var set of int` and the rest are as above.

Example variable choice annotations are: `input_order` choose in order from the array, `first_fail` choose the variable with the smallest domain size, and `smallest` choose the variable with the smallest value in its domain.
Example ways to constraint a variable are: `indomain_min` assign the variable its smallest domain value, `indomain_median` assign the variable its median domain value, `indomain_random` assign the variable a random value from its domain, and `indomain_split` bisect the variables domain excluding the upper half.
Strategy is almost always `complete` for complete search. For a complete list of variable and constraint choice annotations see the FlatZinc specification in the MiniZinc reference documentation.

We can construct more complex search strategies using search constructor annotations. There is only one such annotation at present.

```
seq_search([ search_ann, ..., search_ann ])
```

The sequential search constructor first undertakes the search given by the first annotation in its list, when all variables in this annotation are fixed it undertakes the second search annotation, etc until all search annotations are complete.

Consider the jobshop scheduling model shown in Figure 35. We could replace the solve item with

```
solve :: seq_search([
            int_search(s, smallest, indomain_min, complete),
            int_search([end], input_order, indomain_min, complete)])
      minimize end
```

which tries to set start times `s` by choosing the job that can start earliest and setting it to that time. When all start times are complete the end time `end` may not be fixed. Hence we set it to its minimal possible value.

6.3 Annotations

Annotations are a first class object in MiniZinc. We can declare new annotations in a model, and declare and assign to annotation variables.

NQUEENS-ANN ≡ [DOWNLOAD]

```
annotation bitdomain(int:nwords);

include "alldifferent.mzn";

int: n;
array [1..n] of var 1..n: q :: bitdomain(n div 32);

constraint alldifferent(q) :: domain;
constraint alldifferent([ q[i] + i | i in 1..n]) :: domain;
constraint alldifferent([ q[i] - i | i in 1..n]) :: domain;

ann: search_ann;

solve :: search_ann satisfy;

output [ if fix(q[j]) == i then "Q" else "." endif ++
         if j == n then "\n" else "" endif | i,j in 1..n]
```

Figure 44: Annotated model for n-queens (`nqueens-ann.mzn`).

Annotations

Annotations have a type `ann`. You can declare an annotation parameter (with optional assignment)

ann : ⟨ident⟩ [= ⟨ann-expr⟩] ;

and assign to an annotation variable just as any other parameter.

Expressions, variable declarations, and `solve` items can all be annotated using the `::` operator. We can declare a new annotation using the annotation item

annotation ⟨annotation-name⟩(⟨arg-def⟩, . . . , ⟨arg-def⟩) ;

The program in Figure 44 illustrates the use of annotation declarations, annotations and annotation variables. We declare a new annotation `bitdomain` which is meant to tell the solver that variables domains should be represented via bit arrays of size `nwords`. The annotation is attached to the declarations of the variables `q`. Each of the `alldifferent` constraints is annotated with the built in annotation `domain` which instructs the solver to use the domain propagating version of `alldifferent` if it has one. An annotation variable `search_ann` is declared and used to define the search strategy. We can give the value to the search strategy in a separate data file.

Example search annotations might be the following (where we imagine each line is in a separate data file)

```
search_ann = int_search(q, input_order, indomain_min, complete);
search_ann = int_search(q, input_order, indomain_median, complete);
search_ann = int_search(q, first_fail, indomain_min, complete);
search_ann = int_search(q, first_fail, indomain_median, complete);
```

The first just tries the queens in order setting them to the minimum value, the second tries the queens variables in order, but sets them to their median value, the third tries the queen variable with smallest domain and sets it to the minimum value, and the final strategy tries the queens variable with smallest domain setting it to its median value.

Different search strategies can make a significant difference in how easy it is to find solutions. A small comparison of the number of choices made to find the first solution of the n-queens problems using the 4 different search strategies is shown in the table below (where — means more than 100,000 choices). Clearly the right search strategy can make a significant difference.

n	input-min	input-median	ff-min	ff-median
10	28	15	16	20
15	248	34	23	15
20	37330	97	114	43
25	7271	846	2637	80
30	—	385	1095	639
35	—	4831	—	240
40	—	—	—	236

GROCERY ☰ [DOWNLOAD]

```
var int: item1;
var int: item2;
var int: item3;
var int: item4;

constraint item1 + item2 + item3 + item4 == 711;
constraint item1 * item2 * item3 * item4 == 711 * 100 * 100 * 100;

constraint         0 < item1 /\ item1 <= item2
           /\ item2 <= item3 /\ item3 <= item4;

solve satisfy;

output ["{", show(item1), ",", show(item2), ",", show(item3), ",",
        show(item4),"}\n"];
```

Figure 45: A model with unbounded variables (`grocery.mzn`).

7 Effective Modelling Practices in MiniZinc

There are almost always multiple ways to model the same problem, some of which generate models which are efficient to solve, and some of which are not. In general it is very hard to tell a priori which models are the most efficient for solving a particular problem, and indeed it may critically depend on the underlying solver used, and search strategy. In this chapter we concentrate on modelling practices that avoid inefficiency in generating models and generated models.

7.1 Variable Bounds

Finite domain propagation engines, which are the principle type of solver targeted by MiniZinc are more effective the tighter the bounds on the variables involved. They can also behave badly with problems which have subexpressions that take large integer values, since they may implicitly limit the size of integer variables.

Note that even models where all variables are bounded, may introduce intermediate expressions that are too large for the solver.

The grocery problem shown in Figure 45 finds 4 items whose prices in dollars add up to 7.11 and multiply up to 7.11. The variables are declared unbounded. Running

```
$ mzn-g12fd grocery.mzn
```

yields

```
=====UNSATISFIABLE=====
% grocery.fzn:11: warning: model inconsistency detected before search.
```

This is because the intermediate expressions in the multiplication are also `var ints` and are given default bounds in the solver $-1,000,000..1,000,000$ and these ranges are too small to hold the values of the intermediate expressions may need to take.

Modifying the model so that the items are declared with tight bounds

```
var 1..711: item1;
var 1..711: item2;
var 1..711: item3;
var 1..711: item4;
```

results in a better model, since now MiniZinc can infer bounds on the intermediate expressions and use these rather than the default bounds. With this modification, executing the model gives

```
{120,125,150,316}
----------
```

Note however that even the improved model may be too difficult for some solvers. Running

```
$ mzn-g12lazy grocery.mzn
```

does not return an answer, since the solver builds a huge representation for the intermediate product variables.

Bounding variables

Always try to use bounded variables in models. When using `let` declarations to introduce new variables, always try to define them with correct and tight bounds. This will make your model more efficient, and avoid the possibility of unexpected overflows. One exception is when you introduce a new variable which is immediately defined as equal to an expression. Usually MiniZinc will be able to infer effective bounds from the expression.

7.2 Unconstrained Variables

Sometimes when modelling it is easier to introduce more variables than actually required to model the problem.

Consider the model for Golomb rulers shown in Figure 46. A Golomb ruler of n marks is one where the absolute differences between any two marks are different. It creates a two dimensional array of difference variables, but only uses those of the form `diff[i,j]` where $i > j$. Running the model as

```
$ mzn-g12fd golomb.mzn -D "n = 4; m = 6;"
```

results in output

GOLOMB ≡ [DOWNLOAD]

```
include "alldifferent.mzn";

int: n; % number of marks on ruler
int: m; % max length of ruler

array[1..n] of var 0..m: mark;
array[1..n,1..n] of var 0..m: diffs;

constraint mark[1] = 0;
constraint forall ( i in 1..n-1 ) ( mark[i] < mark[i+1] );
constraint forall (i,j in 1..n where i > j)          % (diff)
                  (diffs[i,j] = mark[i] - mark[j]);  % (diff)
constraint alldifferent([ diffs[i,j] | i,j in 1..n where i > j]);
constraint diffs[2,1] < diffs[n,n-1];  % symmetry break

solve satisfy;

output ["mark = \(mark);\ndiffs = \(diffs);\n"];
```

Figure 46: A model for Golomb rulers with unconstrained variables (`golomb.mzn`).

```
mark = [0, 1, 4, 6];
diffs = [0, 0, 0, 0, 1, 0, 0, 0, 4, 3, 0, 0, 6, 5, 2, 0];
----------
```

and everything seems fine with the model. But if we ask for all solutions using

```
$ mzn-g12fd -a golomb.mzn -D "n = 4; m = 6;"
```

we are presented with a never ending list of the same solution!

What is going on? In order for the finite domain solver to finish it needs to fix all variables, including the variables `diff[i,j]` where $i \leq j$, which means there are countless ways of generating a solution, simply by changing these variables to take arbitrary values.

We can avoid problems with unconstrained variables, by modifying the model so that they are fixed to some value. For example replacing the lines marked `(diff)` in Figure 46 to

```
constraint forall(i,j in 1..n)
                 (diffs[i,j] = if (i > j) then mark[i] - mark[j]
                               else 0 endif);
```

ensures that the extra variables are all fixed to 0. With this change the solver returns just one solution.

MiniZinc will automatically remove variables which are unconstrained and not used in the output. An alternate solution to the above problem is simply to remove the output of the `diffs` array by changing the output statement to

```
output ["mark = \(mark);\n"];
```

With this change running

```
$ mzn-g12fd -a golomb.mzn -D "n = 4; m = 6;"
```

simply results in

```
mark = [0, 1, 4, 6];
----------
==========
```

illustrating the unique solution.

Unconstrained Variables
Models should never have unconstrained variables. Sometimes it is difficult to model without unnecessary variables. If this is the case add constraints to fix the unnecessary variables, so they cannot influence the solving.

7.3 Effective Generators

Imagine we want to count the number of triangles (K_3 subgraphs) appearing in a graph. Suppose the graph is defined by an adjacency matrix: $adj[i,j]$ is true if nodes i and j are adjacent. We might write

```
int: count = sum ([ 1 | i,j,k in NODES where i < j  /\ j < k
                        /\ adj[i,j] /\ adj[i,k] /\ adj[j,k]]);
```

which is certainly correct, but it examines all triples of nodes. If the graph is sparse we can do better by realising that some tests can be applied as soon as we select i and j.

```
int: count = sum( i,j in NODES where i < j /\ adj[i,j])
     (sum([1 | k in NODES where j < k /\ adj[i,k] /\ adj[j,k]]));
```

You can use the builitin `trace` function to help determine what is happening inside generators.

Tracing
The function `trace`(s,e) prints the string s before evaluating the expression e and returning its value. It can be used in any context.

For example, we can see how many times the test is performed in the inner loop for both versions of the calculation.

COUNT1 ≡ [DOWNLOAD]

```
int:count=sum([ 1 | i,j,k in NODES where
          trace("+", i<j /\j<k /\ adj[i,j] /\ adj[i,k] /\ adj[j,k]) ]);
adj = [| false, true,  true,  false
       | true,  false, true,  false
       | true,  true,  false, true
       | false, false, true,  false |];
constraint trace("\n",true);
solve satisfy;
```

Produces the output:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
----------
```

indicating the inner loop is evaluated 64 times while

COUNT2 ≡ [DOWNLOAD]

```
int: count = sum( i,j in NODES where i < j /\ adj[i,j])
  (sum([1 | k in NODES where trace("+", j < k /\ adj[i,k] /\ adj[j,k])]));
```

Produces the output:

```
++++++++++++++++
----------
```

indicating the inner loop is evaluated 16 times.

Note that you can use the dependent strings in `trace` to understand what is happening during model creation.

COUNT3 ≡ [DOWNLOAD]

```
int: count = sum( i,j in NODES where i < j /\ adj[i,j])(
       sum([trace("("++show(i)++","++show(j)++","++show(k)++")",1) |
            k in NODES where  j < k /\ adj[i,k] /\ adj[j,k]]));
```

will print out each of triangles that is found in the calculation. It produces the output

```
(1,2,3)
----------
```

7.4 Redundant Constraints

The form of a model will affect how well the constraint solver can solve it. In many cases adding constraints which are redundant, i.e. are logically implied by the existing model, may improve the search for solutions by making more information available to the solver earlier.

Consider the magic series problem from Section subsection 3.5. Running this for $n = 16$ as follows:

MAGIC-SERIES2 ≡ [DOWNLOAD]

```
int: n;
array[0..n-1] of var 0..n: s;

constraint forall(i in 0..n-1) (
   s[i] = (sum(j in 0..n-1)(bool2int(s[j]=i))));
▶ REDUNDANT
solve satisfy;

output [ "s = ", show(s), ";\n" ] ;
```

Figure 47: Model solving the magic series problem with redundant constraints (`magic-series2.mzn`).

```
$ mzn-g12fd --all-solutions --statistics magic-series.mzn -D "n=16;"
```

might result in output

```
s = [12, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0];
----------
==========
```

and the statistics showing 174 choice points required.

We can add redundant constraints to the model. Since each number in the sequence counts the number of occurrences of a number we know that they sum up to n. Similarly we know that the sum of $s[i] \times i$ must also add up to n because the sequence is magic. Adding these constraints to our model using

REDUNDANT ≡

```
constraint sum(i in 0..n-1)(s[i]) = n;
constraint sum(i in 0..n-1)(s[i] * i) = n;
```

gives the model in Figure 47.

Running the same problem as before

```
$ mzn-g12fd --all-solutions --statistics magic-series2.mzn -D "n=16;"
```

results in the same output, but with statistics showing just 13 choicepoints explored. The redundant constraints have allowed the solver to prune the search much earlier.

7.5 Modelling Choices

There are many ways to model the same problem in MiniZinc, although some may be more natural than others. Different models may have very different efficiency of solving, and worse yet, different models may be better or worse for different solving backends. There are however some guidelines for usually producing better models:

ALLINTERVAL ≡ [DOWNLOAD]

```
include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: x;      % sequence of numbers
array[1..n-1] of var 1..n-1: u;  % sequence of differences

constraint alldifferent(x);
constraint alldifferent(u);
constraint forall(i in 1..n-1)(u[i] = abs(x[i+1] - x[i]));

solve :: int_search(x, first_fail, indomain_min, complete)
      satisfy;
output ["x = ",show(x),"\n"];
```

Figure 48: A natural model for the all interval series problem "prob007" in CSPlib. (`allinterval.mzn`).

Choosing between models

The better model is likely to have some of the following features

- smaller number of variables, or at least those that are not functionally defined by other variables
- smaller domain sizes of variables
- more succinct, or direct, definition of the constraints of the model
- uses global constraints as much as possible

In reality all this has to be tempered by how effective the search is for the model. Usually the effectiveness of search is hard to judge except by experimentation.

Consider the problem of finding permutations of n numbers from 1 to n such that the differences between adjacent numbers also form a permutation of numbers 1 to $n-1$. Note that the u variables are functionally defined by the x variables so the raw search space is n^n The obvious way to model this problem is shown in Figure 48

In this model the array x represents the permutation of the n numbers and the constraints are naturally represented using `alldifferent`. Running the model

```
$ mzn-g12fd -all-solutions --statistics allinterval.mzn -D "n=10;"
```

finds all solutions in 84598 choice points and 3s.

An alternate model uses array y where $y[i]$ gives the position of the number i in the sequence. We also model the positions of the differences using variables v. $v[i]$ is the position

ALLINTERVAL2 ≡ [DOWNLOAD]

```
include "alldifferent.mzn";

int: n;

array[1..n] of var 1..n: y;  % position of each number
array[1..n-1] of var 1..n-1: v; % position of difference i

constraint alldifferent(y);
constraint alldifferent(v);
constraint forall(i,j in 1..n where i < j)(
      (y[i] - y[j] = 1 -> v[j-i] = y[j]) /\
                (y[j] - y[i] = 1 -> v[j-i] = y[i])
    );

constraint abs(y[1] - y[n]) = 1 /\ v[n-1] = min(y[1], y[n]);

solve :: int_search(y, first_fail, indomain_min, complete)
      satisfy;

output [ "x = [",] ++
       [ show(i) ++ if j == n then "]\n;" else ", " endif
         | j in 1..n, i in 1..n where j == fix(y[i]) ];
```

Figure 49: An inverse model for the all interval series problem "prob007" in CSPlib. (`allinterval2.mzn`).

in the sequence where the absolute difference i occurs. If the values of $y[i]$ and $y[j]$ differ by one where $j > i$, meaning the positions are adjacent, then $v[j-i]$ is constrained to be the earliest of these positions. We can add two redundant constraints to this model: since we know that a difference of $n-1$ must result, we know that the positions of 1 and n must be adjacent $|y[1]-y[n]| = 1$, which also tell us that the position of difference $n-1$ is the earlier of $y[1]$ and $y[n]$, i.e. $v[n-1] = min(y[1], y[n])$. With this we can model the problem as shown in Figure 49. The output statement recreates the original sequence x from the array of positions y.

The inverse model has the same size as the original model, in terms of number of variables and domain sizes. But the inverse model has a much more indirect way of modelling the relationship between y and v variables as opposed to the relationship between x and u variables. Hence we might expect the original model to be better.

The command

```
$ mzn-g12fd --all-solutions --statistics allinterval2.mzn -D "n=10;"
```

ALLINTERVAL3 ≡ [DOWNLOAD]

```
include "inverse.mzn";

int: n;

array[1..n] of var 1..n: x;  % sequence of numbers
array[1..n-1] of var 1..n-1: u;  % sequence of differences

constraint forall(i in 1..n-1)(u[i] = abs(x[i+1] - x[i]));

array[1..n] of var 1..n: y;  % position of each number
array[1..n-1] of var 1..n-1: v; % position of difference i

constraint inverse(x,y);
constraint inverse(u,v);

constraint abs(y[1] - y[n]) = 1 /\ v[n-1] = min(y[1], y[n]);

solve :: int_search(y, first_fail, indomain_min, complete)
      satisfy;

output ["x = ",show(x),"\n"];
```

Figure 50: A dual model for the all interval series problem "prob007" in CSPlib. (`allinterval3.mzn`).

finds all the solutions in 75536 choice points and 18s. Interestingly, although the model is not as succinct here, the search on the y variables is better than searching on the x variables. The lack of succinctness means that even though the search requires less choice it is substantially slower.

7.6 Multiple Modelling and Channels

When we have two models for the same problem it may be useful to use both models together by tying the variables in the two models together, since each can give different information to the solver.

Figure 50 gives a dual model combining features of `allinterval.mzn` and `allinterval2.mzn`. The beginning of the model is taken from `allinterval.mzn`. We then introduce the y and v variables from `allinterval2.mzn`. We tie the variables together using the global `inverse`: $inverse(x, y)$ holds if y is the inverse function of x (and vice versa) that is $x[i] = j \Leftrightarrow y[j] = i$. A definition is shown in Figure 51. The model does

INVERSE ☰ [DOWNLOAD]

```
predicate inverse(array[int] of var int: f,
                  array[int] of var int: invf) =
    forall(j in index_set(invf))(invf[j] in index_set(f)) /\
    forall(i in index_set(f))(
        f[i] in index_set(invf) /\
        forall(j in index_set(invf))(j == f[i] <-> i == invf[j])
    );
```

Figure 51: A definition of the `inverse` global constraint. (`inverse.mzn`).

not include the constraints relating the y and v variables, they are redundant (and indeed propagation redundant) so they do not add information for a propagation solver. The `alldifferent` constraints are also missing since they are made redundant (and propagation redundant) by the inverse constraints. The only constraints are the relationships of the x and u variables and the redundant constraints on y and v.

One of the benefits of the dual model is that there is more scope for defining different search strategies. Running the dual model,

```
$ mzn-g12fd -all-solutions --statistics allinterval3.mzn -D "n=10;"
```

which note uses the search strategy of the inverse model, labelling the y variables, finds all solutions in 1714 choice points and 0.5s. Note that running the same model with labelling on x variables requires 13142 choice points and 1.5s.

8 Boolean Satisfiability Modelling in MiniZinc

MiniZinc can be used to model Boolean satisfiability problems where the variables are restricted to be Boolean (`bool`). MiniZinc can be used with efficient Boolean satisfiability solvers to solve the resulting models efficiently.

8.1 Modelling Integers

Many times although we wish to use a Boolean satisfiability solver we may need to model some integer parts of our problem.

There are three common ways of modelling an integer variables I in the range $0..m$ where $m = 2^k - 1$ using Boolean variables.

- Binary: I is represented by k binary variables $i_0, \ldots, i_{k-1}$ where $I = 2^{k-1}i_{k-1} + 2^{k-2}i_{k-2} + \cdots + 2i_1 + i_0$. This can be represented in MiniZinc as

```
array[0..k-1]  of var bool: i;
var 0..pow(2,k)-1: I = sum(j in 0..k-1)(bool2int(i[j])*pow(2,j));
```

- Unary: where I is represented by m binary variables $i_1, \ldots, i_m$ and $i = \sum_{j=1}^{m} bool2int(i_j)$. Since there is massive redundancy in the unary representation we usually require that $i_j \rightarrow i_{j-1}, 1 < j \leq m$. This can be represented in MiniZinc as

```
array[1..m]  of var bool: i;
constraint forall(j in 2..m)(i[j] -> i[j-1]);
var 0..m: I = sum(j in 1..m)(bool2int(i[j]);
```

- Value: where I is represented by $m+1$ binary variables $i_0, \ldots, i_m$ where $i = k \iff i_k$, and at most one of $i_0, \ldots, i_m$ is true. This can be represented in MiniZinc as

```
array[0..m]  of var bool: i;
constraint sum(j in 0..m)(bool2int(i[j]) == 1;
var 0..m: I;
constraint foall(j in 0..m)(I == j <-> i[j]);
```

There are advantages and disadvantages to each representation. It depends on what operations on integers are to required in the model as to which is preferable.

LATIN ≡ [DOWNLOAD]

```
int: n; % size of latin square
array[1..n,1..n] of var 1..n: a;

include "alldifferent.mzn";
constraint forall(i in 1..n)(
             alldifferent(j in 1..n)(a[i,j]) /\
             alldifferent(j in 1..n)(a[j,i])
           );
solve satisfy;
output [ show(a[i,j]) ++ if j == n then "\n" else " " endif |
         i in 1..n, j in 1..n ];
```

Figure 52: Integer Model for Latin Squares (`latin.mzn`).

8.2 Modelling Disequality

Let us considering modelling a latin squares problem. A latin square is an $n \times n$ grid of numbers from $1..n$ such that each number appears exactly once in every row and column. An integer model for latin squares is shown in Figure 52.

The only constraint on the integers is in fact disequality, which is encoded in the `alldifferent` constraint. The value representation is the best way of representing disequality. A Boolean only model for latin squares is shown in Figure 53. Note each integer array element $a[i,j]$ is replaced by an array of Booleans. We use the `exactlyone` predicate to encode that each value is used exactly once in every row and every column, as well as to encode that exactly one of the Booleans corresponding to integer array element $a[i,j]$ is true.

8.3 Modelling Cardinality

Let us consider modelling the Light Up puzzle. The puzzle consists of a rectangular grid of squares which are blank, or filled. Every filled square may contain a number from 1 to 4, or may have no number. The aim is to place lights in the blank squares so that

- Each blank square is "illuminated", that is can see a light through an uninterupted line of blank squares
- No two lights can see each other
- The number of lights adjacent to a numbered filled square is exactly the number in the filled square.

An example of a Light Up puzzle is shown in Figure 54 together with its solution.

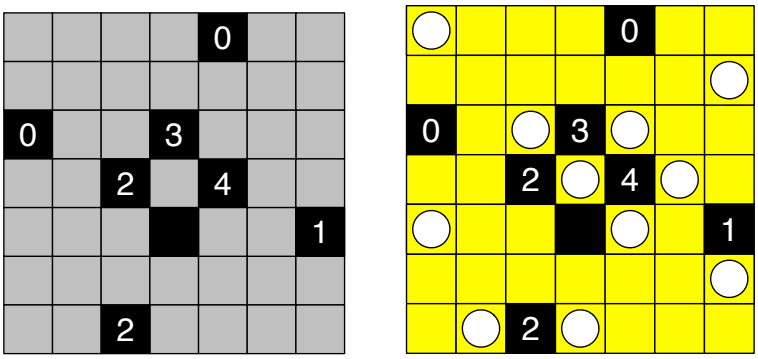
LATINBOOL ≡ [DOWNLOAD]

```
int: n; % size of latin square
array[1..n,1..n,1..n] of var bool: a;

predicate atmostone(array[int] of var bool:x) =
          forall(i,j in index_set(x) where i < j)(
            (not x[i] \/ not x[j]));
predicate exactlyone(array[int] of var bool:x) =
          atmostone(x) /\ exists(x);

constraint forall(i,j in 1..n)(
             exactlyone(k in 1..n)(a[i,j,k]) /\
             exactlyone(k in 1..n)(a[i,k,j]) /\
             exactlyone(k in 1..n)(a[k,i,j])
           );
solve satisfy;
output [ if fix(a[i,j,k]) then
            show(k) ++ if j == n then "\n" else " " endif
         else "" endif | i,j,k in 1..n ];
```

Figure 53: Boolean Model for Latin Squares (`latinbool.mzn`).



(a) (b)

Figure 54: An example of a Light Up puzzle showing (a) the initial puzzle and (b) the completed solution

It is natural to model this problem using Boolean variables to determine which squares contain a light and which do not, but there is some integer arithmetic to consider for the filled squares.

A model for the problem is given in Figure 55. A data file for the problem shown in

LIGHTUP ≡ [DOWNLOAD]

```
int: h; set of int: H = 1..h; % board height
int: w; set of int: W = 1..w; % board width
array[H,W] of -1..5: b;       % board
int: E = -1;                  % empty square
set of int: N = 0..4;         % filled and numbered square
int: F =  5;                  % filled unnumbered square

% position (i1,j1) is visible to (i2,j2)
test visible(int: i1, int: j1, int: i2, int: j2) =
    ((i1 == i2) /\ forall(j in min(j1,j2)..max(j1,j2))(b[i1,j] == E))
 \/ ((j1 == j2) /\ forall(i in min(i1,i2)..max(i1,i2))(b[i,j1] == E));

array[H,W] of var bool: l; % is there a light

% filled squares have no lights
constraint forall(i in H, j in W, where b[i,j] != E)(l[i,j] == false);
% lights next to filled numbered square agree
include "boolsum.mzn";
constraint forall(i in H, j in W where b[i,j] in N)(
    bool_sum_eq([ l[i1,j1] | i1 in i-1..i+1, j1 in j-1..j+1 where
                             abs(i1 - i) + abs(j1 - j) == 1 /\
                             i1 in H /\ j1 in W ], b[i,j]));
% each blank square is illuminated
constraint forall(i in H, j in W where b[i,j] == E)(
             exists(j1 in W where visible(i,j,i,j1))(l[i,j1]) \/
             exists(i1 in H where visible(i,j,i1,j))(l[i1,j])
           );
% no two lights see each other
constraint forall(i1,i2 in H, j1,j2 in W where
                  (i1 != i2 \/ j1 != j2) /\ b[i1,j1] == E
                  /\ b[i2,j2] == E /\ visible(i1,j1,i2,j2))(
             not l[i1,j1] \/ not l[i2,j2]
           );

solve satisfy;
output [ if b[i,j] != E then show(b[i,j])
         else if fix(l[i,j]) then "L" else "." endif
         endif ++ if j == w then "\n" else " " endif |
         i in H, j in W];
```

Figure 55: SAT Model for the Light Up puzzle (`lightup.mzn`).

LIGHTUP.DZN ≡ [DOWNLOAD]

```
h = 7;
w = 7;
b = [| -1,-1,-1,-1, 0,-1,-1
     | -1,-1,-1,-1,-1,-1,-1
     |  0,-1,-1, 3,-1,-1,-1
     | -1,-1, 2,-1, 4,-1,-1
     | -1,-1,-1, 5,-1,-1, 1
     | -1,-1,-1,-1,-1,-1,-1
     |  1,-1, 2,-1,-1,-1,-1 |];
```

Figure 56: Datafile for the Light Up puzzle instance shown in Figure 54

Figure 54 is shown in Figure 56.

The model makes use of a Boolean sum predicate

```
predicate bool_sum_eq(array[int] of var bool:x, int:s);
```

which requires that the sum of an array of Boolean equals some fixed integer. There are a number of ways of modelling such *cardinality* constraints using Booleans.

- Adder networks: we can use a network of adders to build a binary Boolean representation of the sum of the Booleans
- Sorting networks: we can use a sorting network to sort the array of Booleans to create a unary representation of the sum of the Booleans
- Binary decision diagrams: we can create a binary decision diagram (BDD) that encodes the cardinality constraint.

We can implement `bool_sum_eq` using binary adder networks using the code shown in Figure 57. The predicate `binary_sum` defined in Figure 58 creates a binary representation of the sum of x by splitting the list into two, summing up each half to create a binary representation and then summing these two binary numbers using `binary_add`. If the list x is odd the last bit is saved to use as a carry in to the binary addition.

We can implement `bool_sum_eq` using unary sorting networks using the code shown in Figure 59. The cardinality constraint is defined by expanding the input x to have length a power of 2, and sorting the resulting bits using an odd-even merge sorting network. The odd-even merge sorter works shown in Figure 60 recursively by splitting the input list in 2, sorting each list and merging the two sorted lists.

We can implement `bool_sum_eq` using binary decision diagrams using the code shown in Figure 61. The cardinality constraint is broken into two cases: either the first element $x[1]$ is *true*, and the sum of the remaining bits is $s-1$, or $x[1]$ is *false* and the sum of the remaining bits is s. For efficiency this relies on common subexpression elimination to avoid creating many equivalent constraints.

BBOOLSUM ≡ [DOWNLOAD]

```
% the sum of booleans x = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
     let { int: c = length(x) } in
     if s < 0 then false
     elseif s == 0 then
          forall(i in 1..c)(x[i] == false)
     elseif s < c then
          let { % cp = number of bits required for representing 0..c
                int: cp = floor(log2(int2float(c))),
                % z is sum of x in binary
                array[0..cp] of var bool:z  } in
          binary_sum(x, z) /\
          % z == s
          forall(i in 0..cp)(z[i] == ((s div pow(2,i)) mod 2 == 1))
     elseif s == c then
          forall(i in 1..c)(x[i] == true)
     else false endif;

include "binarysum.mzn";
```

Figure 57: Cardinality constraints by binary adder networks (`bboolsum.mzn`).

BINARYSUM ☰ [DOWNLOAD]

```
% the sum of bits x = s in binary.
%            s[0], s[1], ···, s[k] where 2^k >= length(x) > 2^(k-1)
predicate binary_sum(array[int] of var bool:x,
                     array[int] of var bool:s)=
     let { int: l = length(x) } in
     if l == 1 then s[0] = x[1]
     elseif l == 2 then
            s[0] = (x[1] xor x[2]) /\ s[1] = (x[1] /\ x[2])
     else let { int: ll = (l div 2),
                array[1..ll] of var bool: f = [ x[i] | i in 1..ll ],
                array[1..ll] of var bool: t = [x[i]| i in ll+1..2*ll],
                var bool: b = if ll*2 == l then false else x[l] endif,
                int: cp = floor(log2(int2float(ll))),
                array[0..cp] of var bool: fs,
                array[0..cp] of var bool: ts } in
                binary_sum(f, fs) /\ binary_sum(t, ts) /\
                binary_add(fs, ts, b, s)
     endif;

% add two binary numbers x, and y and carry in bit ci to get binary s
predicate binary_add(array[int] of var bool: x,
                     array[int] of var bool: y,
                     var bool: ci,
                     array[int] of var bool: s) =
    let { int:l = length(x),
          int:n = length(s), } in
    assert(l == length(y),
          "length of binary_add input args must be same",
    assert(n == l \/ n == l+1, "length of binary_add output " ++
                        "must be equal or one more than inputs",
    let { array[0..l] of var bool: c } in
    full_adder(x[0], y[0], ci, s[0], c[0]) /\
    forall(i in 1..l)(full_adder(x[i], y[i], c[i-1], s[i], c[i])) /\
    if n > l then s[n] = c[l] else c[l] == false endif ));

predicate full_adder(var bool: x, var bool: y, var bool: ci,
                     var bool: s, var bool: co) =
          let { var bool: xy = x xor y } in
          s = (xy xor ci) /\ co = ((x /\ y) \/ (ci /\ xy));
```

Figure 58: Code for building binary addition networks (`binarysum.mzn`).

UBOOLSUM ≡ [DOWNLOAD]

```
% the sum of booleans x = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
     let { int: c = length(x) } in
     if s < 0 then false
     elseif s == 0 then forall(i in 1..c)(x[i] == false)
     elseif s < c then
          let { % cp = nearest power of 2 >= c
                int: cp = pow(2,ceil(log2(int2float(c)))),
                array[1..cp] of var bool:y, % y is padded version of x
                array[1..cp] of var bool:z  } in
          forall(i in 1..c)(y[i] == x[i]) /\
          forall(i in c+1..cp)(y[i] == false) /\
          oesort(y, z) /\ z[s] == true /\ z[s+1] == false
     elseif s == c then forall(i in 1..c)(x[i] == true)
     else false endif;

include "oesort.mzn";
```

Figure 59: Cardinality constraints by sorting networks (`uboolsum.mzn`).

OESORT ≡ [DOWNLOAD]

```
%% odd-even sort
%% y is the sorted version of x, all trues before falses
predicate oesort(array[int] of var bool:x, array[int] of var bool:y)=
  let { int: c = card(index_set(x)) } in
  if c == 1 then x[1] == y[1]
  elseif c == 2 then comparator(x[1],x[2],y[1],y[2])
  else
    let {
      array[1..c div 2] of var bool:xf = [x[i] | i in 1..c div 2],
      array[1..c div 2] of var bool:xl = [x[i] | i in c div 2 +1..c],
      array[1..c div 2] of var bool:tf,
      array[1..c div 2] of var bool:tl } in
    oesort(xf,tf) /\ oesort(xl,tl) /\ oemerge(tf ++ tl, y)
  endif;

%% odd-even merge
%% y is the sorted version of x, all trues before falses
%% assumes first half of x is sorted, and second half of x
predicate oemerge(array[int] of var bool:x, array[int] of var bool:y)=
  let { int: c = card(index_set(x)) } in
  if c == 1 then x[1] == y[1]
  elseif c == 2 then comparator(x[1],x[2],y[1],y[2])
  else
    let { array[1..c div 2] of var bool:xo =
            [ x[i] | i in 1..c where i mod 2 == 1],
          array[1..c div 2] of var bool:xe =
            [ x[i] | i in 1..c where i mod 2 == 0],
          array[1..c div 2] of var bool:to,
          array[1..c div 2] of var bool:te } in
      oemerge(xo,to) /\ oemerge(xe,te) /\
      y[1] = to[1] /\
      forall(i in 1..c div 2 -1)(
        comparator(te[i],to[i+1],y[2*i],y[2*i+1])) /\
      y[c] = te[c div 2]
  endif));

% comparator o1 = max(i1,i2), o2 = min(i1,i2)
predicate comparator(var bool:i1,var bool:i2,var bool:o1,var bool:o2)=
          (o1 = (i1 \/ i2)) /\ (o2 = (i1 /\ i2));
```

Figure 60: Odd-even merge sorting networks (`oesort.mzn`).

BDDSUM ≡ [DOWNLOAD]

```
% the sum of booleans x = s
predicate bool_sum_eq(array[int] of var bool:x, int:s) =
     let { int: c = length(x),
           array[1..c] of var bool: y = [x[i] | i in index_set(x)]
     } in
     rec_bool_sum_eq(y, 1, s);

predicate rec_bool_sum_eq(array[int] of var bool:x, int: f, int:s) =
     let { int: c = length(x) } in
     if s < 0 then false
     elseif s == 0 then
          forall(i in f..c)(x[i] == false)
     elseif s < c - f + 1 then
          (x[f] == true /\ rec_bool_sum_eq(x,f+1,s-1)) \/
          (x[f] == false /\ rec_bool_sum_eq(x,f+1,s))
     elseif s == c - f + 1 then
          forall(i in f..c)(x[i] == true)
     else false endif;
```

Figure 61: Cardinality constraints by binary decision diagrams (`bddsum.mzn`).

A MiniZinc Keywords

Note that since MiniZinc shares a parser with Zinc, all the Zinc keywords are also not usable as MiniZinc identifiers. The keywords are:

ann, annotation, any, array, assert, bool, constraint, enum, float, function, in, include, int, list, of, op, output, minimize, maximize, par, predicate, record, set, solve, string, test, tuple, type, var, where.

B MiniZinc Operators

The unary operators are: not, + and -. The binary operators are: <->, ->, <-, \/, xor, /\, <, >, <=, >=, ==, =, !=, in, subset, superset, union, diff, symdiff, .., intersect, ++, +, -, *, /, div and mod.

C MiniZinc Functions

The built-in functions in MiniZinc are: abort, abs, acos acosh, array_intersect, array_union, array1d, array2d, array3d, array4d, array5d, array6d, asin, asinh, assert, atan, atanh, bool2int, card, ceil, concat, cos, cosh, dom, dom_array, dom_size, fix, exp, floor, index_set, index_set_1of2, index_set_2of2, index_set_1of3, index_set_2of3, index_set_3of3, int2float, is_fixed, join, lb, lb_array, length, ln, log, log2, log10, min, max, pow, product, round, set2array, show, show_int, show_float, sin, sinh, sqrt, sum, tan, tanh, trace, ub, and ub_array.

Index