Arrays, Sets, Comprehensions

Peter Stuckey

Production Planning Example

A problem with the ToyProblem model is that the production rules and the available resources are hard wired into the model.

It is an example of simple kind of production planning problem in which we wish to

- determine how much of each kind of product to make to maximize the profit where
- manufacturing a product consumes varying amounts of some fixed resources.

- We can use a generic MiniZinc model to handle this kind of problem.

Production Planning Data

```
% Number of different products
int: nproducts;
set of int: PRODUCT = 1..nproducts;

% Profit per unit for each product
array[PRODUCT] of float: profit;

% Number of resources
int: nresources;
set of int: RESOURCE= 1..nresources;

% Amount of each resource available
array[RESOURCE] of float: capacity;

% Units of each resource required to produce
%       1 unit of product
array[PRODUCT,RESOURCE] of float: consumption;
```

Production Planning Constraints

```
% Variables: how much should we make of each product
array[PRODUCT] of var float: produce;

% Must produce a non-negative amount
constraint forall(p in PRODUCT)
                 (produce[p] >= 0.0);


% Production can only use the available resources:
constraint forall (r in RESOURCE)(
 sum (p in PRODUCT)(consumption[p, r] * produce[p])
 <= capacity[r]
);

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

output [ show(produce) ];
```

Production Planning Examples

- ToyProblem

```
nproducts = 2;
profit = [25.0,30.0];
nresources = 1; % hours
capacity = [40.0];
consumption = [| 1.0/200.0 | 1.0/140.0 |];
```

- CakeBaking

```
nproducts = 2;  % banana and chocolate cakes
profit = [4.0, 4.5];
nresources = 5; % flour, banana, sugar, butter, cocoa
capacity = [4000.0, 6.0, 2000.0, 500.0, 500.0];

consumption= [| 250.0, 2.0,  75.0, 100.0,  0.0
              | 200.0, 0.0, 150.0, 150.0, 75.0 |];
```

Sets

Sets are declared by

`set of` *type*

They may be sets of integers, floats or Booleans.

Set expressions:

- Set literals are of form `{e1,…,en}`
- Integer or float ranges are also sets
- Standard set operators are provided: `in, union, intersect, subset, superset, diff, symdiff`
- The size of the set is given by `card`

Some examples:

```
set of int: PRODUCT= 1..nproducts;
{1,2} union {3,4}
```

Sets can be used as types.

Arrays

An array can be multi-dimensional. It is declared by

`array[`*index_set1*`,`*index_set 2*`, …, ] of` *type*

The index set of an array needs to be

- an integer range or
- a fixed set expression whose value is an integer range.

The elements in an array can be anything except another array, e.g.

```
array[PRODUCT, RESOURCE] of int: consume;
array[PRODUCTS] of var 0..mproducts: produce;
```

The built-in function `length` returns the number of elements in a 1-D array

Arrays (Cont.)

1-D arrays are initialized using a list

```
profit = [400, 450];
capacity = [4000, 6, 2000, 500, 500];
```

2-D array initialization uses a list with ``|" separating rows

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

Arrays of any dimension (well ≤ 3) can be initialized from a list using the `arrayⁿd` family of functions:

```
consumption= array2d(1..2,1..5,
 [250,2,75,100,0,200,0,150,150,75]);
```

The concatenation operator ++ can be used with 1-D arrays: `profit = [400]++[450];`

Array & Set Comprehensions

MiniZinc provides comprehensions (like ML)

A set comprehension has form

{ *expr* | *generator1*, *generator2*, … }

{ *expr* | *generator1*, *generator2*, … `where` *bool-expr* }

An array comprehension is similar

[*expr* | *generator1*, *generator2*, …]

[*expr* | *generator1*, *generator2*, … `where` *bool-expr*]

E.g. `{i + j | i, j in 1..4 where i < j}`

```
= {1 + 2, 1 + 3, 1 + 4, 2 + 3, 2 + 4, 3 + 4}
= {3, 4, 5, 6, 7}
```

Array & Set Comprehensions Question

Exercise: What does b =?

```
set of int: COL = 1..5;
set of int: ROW = 1..2;
array[ROW,COL] of int: c =
    [| 250, 2,  75, 100,  0
     | 200, 0, 150, 150, 75 |];
b = array2d(COL, ROW,
   [c[j, i] | i in COL, j in ROW]);
```

Array & Set Comprehension Answer

- b is the transpose of c

```
[c[j, i] | i in COL, j in ROW ] =
[ 250, 200, 2, 0, 75,
  150, 100, 150, 0, 75]

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

Iteration

MiniZinc provides a variety of built-in functions for operating over a list or set:

- Lists of numbers: `sum, product, min, max`
- Lists of constraints: `forall, exists`

MiniZinc provides a special syntax for calls to these (and other generator functions)

For example,

```
forall (i, j in 1..10 where i < j)
       (a[i] != a[j]);
```

is equivalent to

```
forall ([ a[i] != a[j]
        | i, j in 1..10 where i < j]);
```

Overview

- Real models apply to different sized data

- MiniZinc uses
  - sets to name objects
  - arrays to capture information about objects

  - comprehensions to build
    - constraints, and
    - expressions

    about different sized data

EOF