Sets, Arrays, Comprehensions

Jimmy Lee & Peter Stuckey

Sets

- Sets are declared by
  - `set of` *type*
- They may be sets of integers, enums, 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`
- Sets can be used as types

2

Set examples

- Examples of some fixed sets

```
set of int: ROW = 1..6;
set of int: PRIME = {2,3,5,7,11,13};
set of float: RAN = 3.0 .. 5.0;
set of float: NUM = {2.87, 3.14};
set of bool: TRUE = {true};
```

- Example set expressions

```
set of int: x = ROW union PRIME;
set of float: y = RAN intersect NUM;
set of int: z = ROW symdiff PRIME;
bool: b = ROW subset PRIME;
set of bool: bs = { PRIME subset ROW,
 false };
```

3

Set examples

- Example set expressions

```
set of int: x = ROW union PRIME;
set of float: y = RAN intersect NUM;
set of int: z = ROW symdiff PRIME;
bool: b = ROW subset PRIME;
set of bool: bs = { PRIME subset ROW,
 false };
```

- Values

```
x = {1,2,3,4,5,6,7,11,13};
y = {3.14};
z = {1,4,6,7,11,13};
b = false;
bs = { false };
```

4

Arrays

- An array can be multi-dimensional. It is declared
  - `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.
  - an enumerated type range expression
- 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: prod;`
- The built-in function `length` returns the number of elements in a 1-D array

5

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 (≤ 6) 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]);
```

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

6

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*]
- A generator is of the form *var* `in` *set-expr*
- An alternative syntax for two (or more) generators using the same set is *v1*, *v2* `in` *set-expr*
- shorthand for *v1* `in` *set-expr*, *v2* `in` *set-expr*

7

Comprehending Comprehension

- [*expr* | *generator1*, *generator2*, … `where` *bool-expr*]
  - give a value to the variable of the *generator1*
  - give a value for the variable of the *generator2*
  - …
  - evaluate the *bool-expr* after where
  - if it is true
    - generate a element *expr*
  - try the next value for the last generator
  - if exhausted try the next value for previous generator
  - until all generators exhausted

8

Comprehending Comprehensions

- For example

```
  { 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}
```

- Example

```
set of int: PRIME = {2,3,5,7,11,13};
array[int] of int: xs =
  [ abs(i - j)
  | i in PRIME, j in i-1..6
    where (i+j) mod 2 = 1 ];

= [ abs(2-1), abs(2-3), abs(2-5),
 abs(3-2), abs(3-4), abs(3-6), abs(5-
 4), abs(5-6), abs(7-6) ]
= [ 1, 1, 3, 1, 1, 3, 1, 1, 1 ];
```

9

Array 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]);
```

10

Array 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
     |];
```

11

Array Comprehension Examples

- What is the value of the following expressions

```
set of int: PRIME = {2,3,5,7,11,13};
x = { 3*i+j | i in 2..3, j in PRIME };
y = { i*j | i in PRIME, j in 2..i
            where i mod j = 0 };
z = [{j | j in 1..i where j mod 2=1 }
    | i in PRIME where i != 11 ];
```

- Values

```
x = {8,9,11,12,13,14,16,17,19,20,22};
y = {4,9,25,49,121,169};
z = [{1}, {1,3}, {1,3,5}, {1,3,5,7},
 {1,3,5,7,9,11,13}];
```

12

Generator Call Syntax

- MiniZinc provides a variety of built-in functions for operating over a 1D array:
  - Lists of numbers: `sum, product, min, max`
  - Lists of constraints: `forall, exists`
- MiniZinc provides a special syntax for calls to any function taking a 1D array

function `(` *generator*`,` *generator* `where` *bexpr* `)`
`(` *expr* `)`

- is shorthand for

function `([` *expr*
`|` *generator*`,` *generator* `where` *bexpr* `])`

13

Generator Call Examples

- 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]);
```

14

Generator Call examples

- What is the value of the expressions

```
set of int: PRIME = {2,3,5,7,11,13};
x = sum(i in 2..3, j in 3..4)(i+j);
u = sum({i+j| i in 2..3, j in 3..4});
y = min(i in PRIME, j in 2..3)
       ( i*j mod 30 );
z = exists(i in PRIME, j in 2..i-1)
          (i mod j = 0);
```

- Value

```
x = 24;     % sum([5,6,6,7])
u = 18;     % sum({5,6,7})
y = 3;      % when i = 11, j = 3
z = false;  % array is all false
```

15

Overview

- MiniZinc uses
  - sets to name sets of objects
  - arrays to capture information about objects
  - comprehensions to build
    - constraints, and
    - expressions

    about different sized data

- You will need to learn to use comprehensions to build the data and constraints you need to model complex problems
- For more on comprehensions see
  - https://en.wikipedia.org/wiki/List_comprehension

16