* If you want to create a **set of value** you use :

**. set of int: “Name of variable”: 1..n**

* for an array with double index:

**. array[Height, width] of var float: t**

**. enum type** is what we use for the strings:

so we can do:

enum Products;

nel datafile Products = {name1, name2,name 3}

. You can create also an array with more than one indexes:

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.

* **array[〈index-set 1 〉, ... , 〈index-set n 〉] of 〈type-inst〉**

[| 〈expr1,1〉, ... , 〈expr1,n 〉, | ..., | 〈exprm,1〉, ... , 〈exprm,n 〉 |]

In this case:

consumption= [| 250, 2, 75, 100, 0, |

200, 0, 150, 150, 75 |];

* **The concatenation operator ‘++’** can be used to concatenate two one-dimensional arrays together.

For instance [4000, 6] **++** [2000, 500, 500] evaluates to [4000, 6, 2000, 500, 500].

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]**

I can initialized a variable also in this way:

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

**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 ⊥ 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 ⊥ if i is less than or equal to 0 or greater then the number of elements in Enum.

• **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.

**• Boolean literals are true and false**,

**• 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**

Another important things is that decision variables can be used for array access.

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

* 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

* 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)

questo verifica che l’array unidimensionale definito a sinistra sia contenuto nell’array a doppio indice.

Cioè verifico che una tupla sia contenuta in un insieme di tuple.

var s .. e: x;

let {int: l = s div 2;

int: u = e div 2;

var l .. u: y;} in x = 2\*y

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) );

If you want to choose a search startegy, in this case the first fail we have to insert this:

solve :: int\_search(q, first\_fail, indomain\_min, complete)

satisfy;

This search annotation means that we should search by selecting from the array of integer variables q.

the variable with the smallest current domain, and try setting it to its smallest possible value), looking

across the entire search tree (complete search).

I can utilize **indomain\_min** which detect the minimun value in the domain

I can utilize the **indomain\_median** which detect the median domain value

I can utilize the **indomain\_random** which can detect a random value in the domain.

**indomain\_split** bisect the domain in two parts and exclude the upper half.

**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.

We can construct more complex search strategies,

solve :: seq\_search([ int\_search(s, smallest, indomain\_min, complete),

int\_search([end], input\_order, indomain\_min, complete)])

minimize end