Il linguaggio MiniZinc

Marco Gavanelli
Dipartimento di Ingegneria

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Linguaggi di Specifica

- Programmazione Logica a Vincoli
- Poi implementata anche in linguaggi non logici (in genere, ad oggetti, soprattutto C++ e Java)
- Molti solver disponibili, ciascuno con il suo linguaggio specifico
- Alcuni di questi solver avevano un linguaggio di specifica; (OPL per ILOG, Mosel per XpressMP, …)
- Proposta: un linguaggio per scrivere modelli CSP e COP che si interfacciasse a molti solver.

Il linguaggio MiniZinc

- È un linguaggio per descrivere problemi di soddisfacimento di vincoli (e problemi di ottimizzazione)
- Permette quindi di definire:
  - Variabili
  - Domini
  - Vincoli
  - Funzione obiettivo

MiniZinc e FlatZinc

- Le specifiche scritte in MiniZinc vengono convertite in linguaggio FlatZinc tramite un convertitore mzn2fzn
- Le specifiche FlatZinc possono poi essere risolte tramite molti solver
  - COIN-OR CBC
  - Gurobi
  - IBM ILOG CPLEX
  - Gecode
  - Chuffed
  - Choco 3
  - ECLiPSe
  - HaifaCSP
  - JaCoP
  - MinisatID
  - Mistral 2.0
  - Opturion CPX
  - OR-Tools
  - OscaR/CBLS
  - Picat
  - SICStus
  - SCIP
  - Yuck

Più alcuni solver sviluppati per MiniZinc, già integrati in un IDE

Per usare ECLiPSe con MiniZinc

- Dalla command-line del sistema operativo

```
mzn2fzn --output-to-stdout model.mzn |
  eclipse -e "flatzinc:fzn_run(fzn_fd)"
```

- oppure da ECLiPSe

```
lib(minizinc).
mzn_run("model.mzn", fzn_fd).
```

- oppure

```
mzn_run("model.mzn", "instance.dzn", fzn_fd).
```

se c'è anche un file di istanza

MiniZinc: Variabili e domini

- Per definire una variabile e il suo dominio

```
var 1..10 : nome;
```

- In questo modo dichiariamo che esiste una variabile chiamata **nome** il cui dominio va da 1 a 10.
- Oppure

```
var {1,7,8,10} : nome;
```

- Oppure:

```
var int : nome;
```

- se non si vuole definire il dominio (Dominio a default –MaxInt…Maxint)

MiniZinc: Vincoli

- Per definire un vincolo, si usa la parola chiave `constraint`
- Ad esempio

```
var 1..10 : x;
var 0..5 : y;
constraint x<=y;
```

MiniZinc: Far partire la ricerca

- Per far partire la ricerca di una soluzione, si usa

```
solve satisfy;
```

- Ad esempio

```
var 1..10 : x;
var 0..5 : y;
constraint x<=y;
solve satisfy;
```

```
Compiling prova.mzn
Running prova.mzn
x = 1;
y = 1;
----------
Finished in 9msec
```

- cerca una qualunque soluzione che soddisfi i vincoli

MiniZinc: Altri Vincoli

- Alcuni vincoli:
  - x > y x è maggiore di y
  - x < y x è minore di y
  - x = y x è uguale a y
  - x != y x è diverso da y
  - x >=y x è maggiore o uguale a y
  - x <=y x è minore o uguale a y
- All'interno dei vincoli si possono usare gli operatori:
  - + somma
  - - sottrazione
  - * prodotto
  - div divisione intera
  - mod resto della divisione intera
  - abs valore assoluto
  - pow(b,e) potenza b^e
- Ad esempio, si può scrivere x*y <= x+2*z – k*k/z

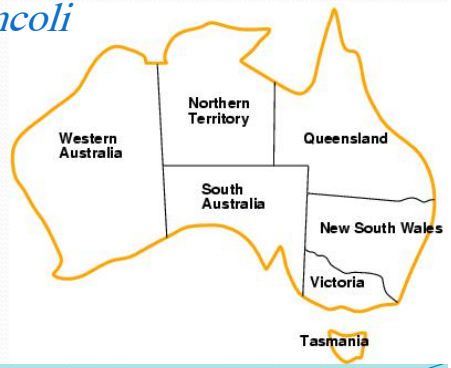
Esempio

```
var 1..3 : WA;
var 1..3 : NT;
var 1..3 : SA;
...
constraint WA != NT;
constraint WA != SA;
constraint NT != SA;
...
solve satisfy;
```

Definizione di variabili e domini

Vincoli

Ricerca



Output

- MiniZinc ha un output a default; se lo si vuole cambiare, si inserisce

  `output` L;

- dove L è una lista di stringhe.
- Le stringhe sono delimitate da doppie virgolette, con le convenzioni C per i caratteri speciali "\n" …
- **`show(e)`** trasforma l'espressione **`e`** in stringa
- concatenazione di stringhe **`++`**
- Es **`output ["t = ", show(t), "\n"];`**
- invece di usare **`show`**, si può usare **`\(e)`** all'interno di una stringa:
  - invece di **`"t=" ++ show(t) ++ "\n"`**
  - si può usare **`"t=\(t)\n"`**

Il linguaggio MiniZinc

Uso dei Data Files

Marco Gavanelli

Dipartimento di Ingegneria

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Data Files

- Spesso accade che si voglia avere un modello e varie istanze, con diversi dati di input
- I file contenenti il modello CSP hanno estensione .mzn
- Nel file modello si possono lasciare non specificati i valori dei parametri:
- es `int : x;`
- I valori dei parametri possono essere forniti in un datafile, con estensione .dzn
- es: `x=12;`

Il linguaggio MiniZinc

Array e liste,
List Comprehensions

Marco Gavanelli
Dipartimento di Ingegneria

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

MiniZinc: Array

- A volte è utile definire una sequenza di variabili

```
array[1..5] of var 3..10 : x;
```

- definisce x come una sequenza di 5 variabili, ciascuna con dominio da 3 a 10.

- Poi posso imporre vincoli su queste come delle normali variabili.
- Ad esempio

```
constraint x[1]<x[2];
```

Array

- In generale, si possono avere array di costanti e di variabili

  array [<index-set$_1$ >, ...,<index-set$_n$>] of <type-inst>
- Si possono scrivere letterali di tipo array

  [expr 1 , ... , expr n]
- e di tipo matrice

  [|<expr $_{1,1}$>, ... ,<expr $_{1,n}$> | … | <expr $_{m,1}$>, ... , <expr $_{m,n}$ >|]
- l'operatore di concatenazione ++ si può usare anche per gli array
- Gli array non possono contenere array

MiniZinc: liste

- Array ad una dimensione con indice che parte da 1 possono essere considerati come liste
- Posso avere liste di valori e di variabili

`[1,5,3]` `[a,b,3]`

- Ad esempio, in matematica posso scrivere

$$\{x_i | i \in 1..4\}$$

- per rappresentare l'insieme

$$\{x_1, x_2, x_3, x_4\}$$

- In MiniZinc posso scrivere

```
[x[i]|i in 1..4]
```

- oppure

```
[x[1],x[2],x[3],x[4]]
```

List Comprehensions

- Forma generale

[expr | generator-exp]

- dove expr specifica come costruire elementi nella lista di output a partire dagli elementi generati da generator-exp
- generator-exp può essere
  - generator , ..., generator
  - generator , ..., generator **where** bool-exp
- generator ha la forma

identifier ,..., identifier in array-exp

- Di solito bool-expr non ha variabili decisionali (invece <expr> sì!)
- Es `[i + j | i, j in 1..3 where j < i]` equivalente a `[1+2, 1+3, 2+3]` ovvero `[3, 4, 5]`

Il linguaggio MiniZinc

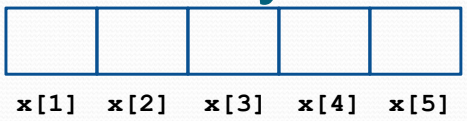
Aggregatori

Marco Gavanelli

Dipartimento di Ingegneria

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

MiniZinc: Array



- Se voglio che i valori siano ordinati

```
constraint x[1]<x[2];
constraint x[2]<x[3];
constraint x[3]<x[4];
constraint x[4]<x[5];
```

- Sarebbe bello scrivere

$$\forall i \in 1..4,\ x_i < x_{i+1}$$

```
constraint forall(i in 1..4)(x[i]<x[i+1]);
```

Aggregatori

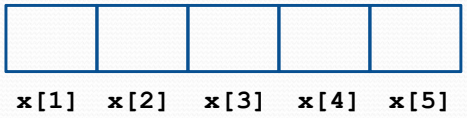
- le funzioni di aggregazione prendono come parametro un array (di costanti o variabili decisionali) e forniscono un valore (costante o variabile decisionale)
- In particolare, ci sono funzioni che prendono array di boolean e restituiscono un boolean:

  `forall( <array di bool> )`

  restituisce l'AND dei boolean
- Nota che i vincoli restituiscono sempre un boolean
- quindi posso imporre una lista di vincoli con

  `constraint forall ( [x<y,x<z,y<z] )`

Esercizio



- Imporre che un array di 5 interi sia costituito da valori in ordine crescente.
- **Esercizio**: imporre che un array abbia valori tutti diversi

Altri aggregatori boolean

- **`exists`**(<array di bool>)
  - fornisce l'OR dei boolean
- **`xorall`**(<array di bool>)
  - XOR
  - impone che un numero dispari di bool sia vero
- **`iffall`**(<array di bool>)
  - XNOR
  - impone che un numero pari di bool sia vero

Aggregatori numerici

- **`sum`**(<array di int>)
  - fornisce la somma. Array vuoto -> 0
- **`product`**(<array di int>)
  - fornisce il prodotto. Array vuoto -> 1
- **`min`**(<array di int>)
  - minimo. Array vuoto -> errore
- **`max`**(<array di int>)
  - massimo. Array vuoto -> errore

Generator call expressions

- Quando si usano gli aggregatori con le comprehension, invece di

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

- si può scrivere

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

- Le due sono del tutto equivalenti (zucchero sintattico).
- In generale,

  agg-func (generator-exp) (exp)
- è equivalente a

  agg-func ([expr | generator-exp])

Il linguaggio MiniZinc

Matrici
Ottimizzazione
Vincoli Globali

Marco Gavanelli
Dipartimento di Ingegneria

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Costanti

- In MiniZinc, si possono definire delle costanti

```
int : N=3;
```

- (invece per le variabili: `var 1..5 : x;`)
- Così posso definire la matrice

```
array[1..N] of var 1..100000 : Q ;
```

Matrici

- Una matrice è un array a due dimensioni

```
array[1..3,1..4] of var 1..5: M;
```

- definisce **M** come una matrice

M[1,1]	M[1,2]	M[1,3]	M[1,4]
M[2,1]	M[2,2]	M[2,3]	M[2,4]
M[3,1]	M[3,2]	M[3,3]	M[3,4]

- Se voglio imporre che la prima riga abbia tutti valori diversi

```
constraint alldifferent([M[1,i]|i in 1..4]);
```

Matrici

- Una matrice è un array a due dimensioni

```
array[1..3,1..4] of var 1..5: M;
```

- definisce **M** come una matrice

M[1,1]	M[1,2]	M[1,3]	M[1,4]
M[2,1]	M[2,2]	M[2,3]	M[2,4]
M[3,1]	M[3,2]	M[3,3]	M[3,4]

- Se voglio imporre che *ogni* riga abbia tutti valori diversi

```
constraint forall(j in 1..3)
  (alldifferent([M[j,i]|i in 1..4]));
```

Ottimizzazione

- In alcuni casi, trovare una soluzione non è sufficiente: si desidera trovare la soluzione *migliore* fra tutte quelle possibili
- Per prima cosa bisogna definire che cosa vuol dire *migliore*
  - la soluzione a costo più basso
  - la soluzione che fa guadagnare di più
  - la soluzione in cui si spreca meno tempo
  - …
- In tutti questi casi, si usa una funzione obiettivo, che in alcuni casi va massimizzata, in altri minimizzata.

Ottimizzazione

- Per richiedere di ottimizzare una funzione obiettivo, invece di **solve satisfy** si usa
- per un problema di massimizzazione

```
solve maximize funzione;
```

- per un problema di minimizzazione

```
solve minimize funzione;
```

- Ad es

```
var 1..5 : x;
var 1..5 : y;
constraint x*x+y*y < 10;
solve maximize 2*x+y;
```

Vincoli Globali

- **alldifferent(array[int] of var int: x)**
- Per usarlo:
- **include "alldifferent.mzn";**
- per i vincoli globali:
- **include "globals.mzn";**
- **cumulative(array[int] of var int: s,**
  array[int] of var int: d,
  array[int] of var int: r, var int: b)

Vincoli globali

- **`table(array[int] of var TIPO: x, array[int, int] of TIPO: t)`**
- Dove TIPO può essere bool o int.
- Impone che **`x`** sia una delle righe della tabella **`t`**.
- Utile per creare nuovi vincoli (simile a **`element`**, ma fa più propagazione)

MiniZinc: Vincoli su liste

- Poi posso imporre vincoli su liste (di variabili o anche di valori)
- **`alldifferent(Lista)`**: impone che tutti i valori nella lista siano diversi
  Ad es
  ```
  constraint alldifferent([x,y,z,3]);
  ```
  impone:
  ```
  x!=y, x!=z, x!=3
  y!=z, y!=3,
  z!=3
  ```
  NOTA: per usare alldifferent, bisogna scrivere all'inizio del programma
  ```
  include "alldifferent.mzn";
  ```
- **`sum(Lista)`**: calcola la somma degli elementi della lista
  Ad esempio
  ```
  constraint sum([x,y,z,3])=s;
  ```
  impone
  ```
  x+y+z+3 = s
  ```

Conditional expressions

- **`if`** *boolexp* **`then`** *exp*1 **`else`** *exp*2 **`endif`**
- è una espressione condizionale. Il risultato è dello stesso tipo di *exp*1 e di *exp*2 (che devono essere dello stesso tipo)
- Se *boolexp* è vera, il risultato è la valutazione di *exp*1, altrimenti è la valutazione di *exp*2.
- Il ramo **`else`** è obbligatorio

Conditional expressions

- **`if`** *boolexp* **`then`** *exp*1 **`else`** *exp*2 **`endif`**
- Può essere usato per imporre vincoli sotto condizioni che dipendono dall'istanza
  ```
  int : a; var 1..5 : x; var 3..6 : y;
  constraint if a>0 then x<y else true
  endif;
  ```
- Per imporre condizioni che necessitano di vincoli reificati
  ```
  var 1..10 : a; var 1..3 : b; var 2..4 : y;
  constraint if a>5 then y=a else y=b endif;
  ```
- Può fornire anche risultati di tipo diverso dal boolean
  ```
  var 1..10 : a; var 1..3 : b; var 2..4 : y;
  constraint y< if a>b then a else b endif;
  ```

Sudoku

- Nel Sudoku, viene dato uno schema 9x9 con alcune celle compilate
- Ci sono anche 9 riquadri 3x3
- Bisogna inserire nelle celle bianche i numeri da 1 a 9 in modo che
- in ogni riga non ci siano due nume uguali
- in ogni colonna non ci siano due numeri uguali
- In ogni riquadro non ci siano due numeri uguali

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

http://www.unife.it/ing/lm.infoauto/constraint-programming/materiale-didattico-2/lucidi/sudoku.dzn