universität
innsbruck

Seminar mit Bachelorarbeit Lehramt

Tobias Hell **Georg Moser**

cbr.uibk.ac.at

Definition (Cryptoarithmetic)

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Definition (Cryptoarithmetic)

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \mathrm{S\ E\ N\ D} \\ \mathrm{M\ O\ R\ E} \\ \hline \mathrm{M\ O\ N\ E\ Y} \end{array} +$$

Definition (Cryptoarithmetic)

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \text{S E N D} \\ \text{M O R E} \\ \hline \text{1 O N E Y} \end{array} +$$

Definition (Cryptoarithmetic)

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \mathrm{SEND} \\ \mathrm{1ORE} \\ \hline \mathrm{1ONEY} \end{array} +$$

Definition (Cryptoarithmetic)

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \mathrm{SEND} \\ \mathrm{10RE} \\ \hline \mathrm{10NEY} \end{array} +$$

Definition (Cryptoarithmetic)

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} 9\,E\,N\,D \\ 1\,0\,R\,E \\ \hline 1\,0\,N\,E\,Y \end{array} +$$

Definition (Cryptoarithmetic)

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} 9\,E\,N\,D \\ 1\,0\,8\,E \\ \hline 1\,0\,N\,E\,Y \end{array} +$$

Definition (Cryptoarithmetic)

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} 9\,5\,N\,D \\ 1\,0\,8\,5 \\ \hline 1\,0\,N\,5\,Y \end{array} +$$

first attempt

```
solve([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]]) :-
        Digits = [D, E, M, N, O, R, S, Y],
        Carries = [C1,C2,C3,C4],
        selects(Digits, [0,1,2,3,4,5,6,7,8,9]),
        members(Carries, [0,1]),
        M           =:=          C4,
        O + 10 * C4 =:= S + M + C3,
        N + 10 * C3 =:= E + O + C2,
        E + 10 * C2 =:= N + R + C1,
        Y + 10 * C1 =:= D + E,
        M > 0, S > 0.

:- solve(X),
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]].
```

very inefficient

```
?- time(solve(X)).
% 133,247,057 inferences,
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

very inefficient

```
?- time(solve(X)).
% 133,247,057 inferences,
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

explanation

- generate-and-test in it’s purest form
- all guesses are performed before the constraints are checked
- arithmetic checks cannot deal with variables

very inefficient

```
?- time(solve(X)).
% 133,247,057 inferences,
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

explanation

- generate-and-test in it’s purest form
- all guesses are performed before the constraints are checked
- arithmetic checks cannot deal with variables

improvement

- move testing into generating
- destroys clean structure of program

Constraint Logic Programming

Definitions (CLP on finite domains)

- `use_module(library(clpfd))` loads the clpfd library
- `Xs ins N .. M` specifies that all values in *Xs* must be in the given range
- `all_different(Xs)` specifies that all values in *Xs* are different
- `label(Xs)` all variables in *Xs* are evaluated to become values
- `#=`, `#\=`, `#>`, . . . like the arithmetic comparison operators, but may contain (constraint) variables

Constraint Logic Programming

Definitions (CLP on finite domains)

- `use_module(library(clpfd))` loads the clpfd library
- `Xs ins N .. M` specifies that all values in *Xs* must be in the given range
- `all_different(Xs)` specifies that all values in *Xs* are different
- `label(Xs)` all variables in *Xs* are evaluated to become values
- `#=`, `#\=`, `#>`, . . . like the arithmetic comparison operators, but may contain (constraint) variables

standard approach

- load the library
- specify all constraints
- call `label` to start efficient computation of solutions

Second Attempt

constraint logic program

```
solve([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]]) :-
      Digits = [D, E, M, N, O, R, S, Y],
      Carries = [C1,C2,C3,C4],
      Digits  ins 0 .. 9, all_different(Digits),
      Carries ins 0 .. 1,
      M           #=          C4,
      O + 10 * C4 #= S + M + C3,
      N + 10 * C3 #= E + O + C2,
      E + 10 * C2 #= N + R + C1,
      Y + 10 * C1 #= D + E,
      M #> 0, S #> 0,
      label(Digits).
```

Definition (Sudoku)

- Sudoku is a well-known logic puzzle; usually played on a 9×9 grid
- $\forall$ *cells*: *cells* $\in \{1, \ldots, 9\}$
- $\forall$ *rows*: all entries are different
- $\forall$ *colums*: all entries are different
- $\forall$ *blocks*: all entries are different

Definition (Sudoku)

- Sudoku is a well-known logic puzzle; usually played on a 9×9 grid
- $\forall$ *cells*: *cells* $\in \{1, \ldots, 9\}$
- $\forall$ *rows*: all entries are different
- $\forall$ *colums*: all entries are different
- $\forall$ *blocks*: all entries are different

Main Loop (using clp)

```
sudoku(Puzzle) :-
        show(Puzzle),
        flatten(Puzzle,Cells),
        Cells ins 1 .. 9,
        rows(Puzzle),
        cols(Puzzle),
        blocks(Puzzle),
        label(Cells),
```

auxiliary predicates

- *flatten*/2 flattens a list
- *show*/1 prints the current puzzle

auxiliary predicates

- *flatten*/2 flattens a list
- *show*/1 prints the current puzzle

row/1

```
rows([]).
rows([R|Rs]) :-
        all_different(R), rows(Rs).
```

auxiliary predicates

- *flatten*/2 flattens a list
- *show*/1 prints the current puzzle

row/1

```
rows([]).
rows([R|Rs]) :-
        all_different(R), rows(Rs).
```

row/1 (alternative)

```
rows(Rs) :- maplist(all_distinct,Rs).
```

cols/1

```
cols([[]|_]).
cols([
      [X1|R1],
      [X2|R2],
      [X3|R3],
      [X4|R4],
      [X5|R5],
      [X6|R6],
      [X7|R7],
      [X8|R8],
      [X9|R9]]) :-
        all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
        cols([R1,R2,R3,R4,R5,R6,R7,R8,R9]).
```

cols/1

```
cols([[]|_]).
cols([
      [X1|R1],
      [X2|R2],
      [X3|R3],
      [X4|R4],
      [X5|R5],
      [X6|R6],
      [X7|R7],
      [X8|R8],
      [X9|R9]]) :-
        all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
        cols([R1,R2,R3,R4,R5,R6,R7,R8,R9]).
```

cols/1 (alternative)

use *maplist*/2

blocks/1

```
blocks([]).
blocks([[],[],[]|Rs]) :- blocks(Rs).
blocks([[X1,X2,X3|R1],
        [X4,X5,X6|R2],
        [X7,X8,X9|R3] |Rs]) :-
        all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
        blocks([R1,R2,R3|Rs]).
```

blocks/1

```
blocks([]).
blocks([[],[],[]|Rs]) :- blocks(Rs).
blocks([[X1,X2,X3|R1],
        [X4,X5,X6|R2],
        [X7,X8,X9|R3] |Rs]) :-
        all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
        blocks([R1,R2,R3|Rs]).
```

Example

```
:- sudoku([[1,_,_,_,_,_,_,_,_],
        [_,_,2,7,4,_,_,_,_],
        [_,_,_,5,_,_,_,_,4],
        [_,3,_,_,_,_,_,_,_],
        [7,5,_,_,_,_,_,_,_],
        [_,_,_,_,_,9,6,_,_],
        [_,4,_,_,_,6,_,_,_],
        [_,_,_,_,_,_,_,7,1],
        [_,_,_,_,_,1,_,3,_]]).
```