Programming Paradigms

Lecture 12

Slides are from Prof. Chin Wei-Ngan from NUS

Relational and Constraint Programming

Reminder of the Last Lecture

- Stateful programming
  - what is state?
  - cells as abstract datatypes
  - the stateful model
  - relationship between the declarative model and the stateful model
  - indexed collections:
    - array model
  - system building
    - component-based programming

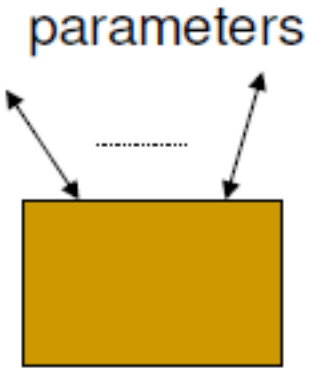
Overview

- Relational Programming
- Choice and Fail Operations
- Constraint Programming
- Basic Constraints, Propagators and Search

Relations

- Functions are *directional* and computes output(s) from inputs.

- Relations are *bidirectional* and used to relate a tuple of parameters.



Examples of Relations

- Parent-child relations e.g. `parent(X,Y)`
- Classification e.g. `male(X)` or `female(Y)`
- Operations e.g. `append(Xs,Ys,Zs)`
- Databases: `employee(Name,… )` relational tables?
- Geometry problems: how are sides of rectangles related, e.g. `rect(X,Y,X,Y)`
- Each function is a special case of relation.

The Relational Model

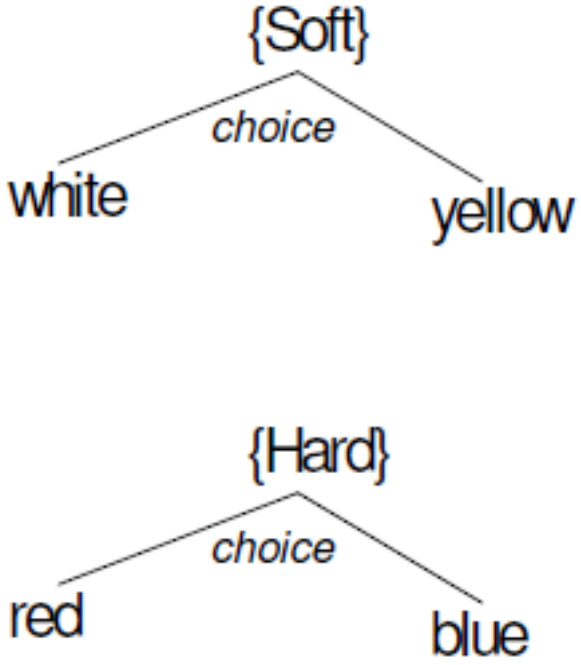
⟨S⟩ ::=	`skip`	*empty statement*
|	⟨S_1⟩ ⟨S_2⟩	*statement sequence*
|	...	
|	`choice` ⟨S_1⟩ `[]` ... `[]` ⟨S_n⟩ `end`	*choice*
|	`fail`	*failure*

Choice allows alternatives to be explored,
while failure indicates no answer at that branch.

Clothing Design Example

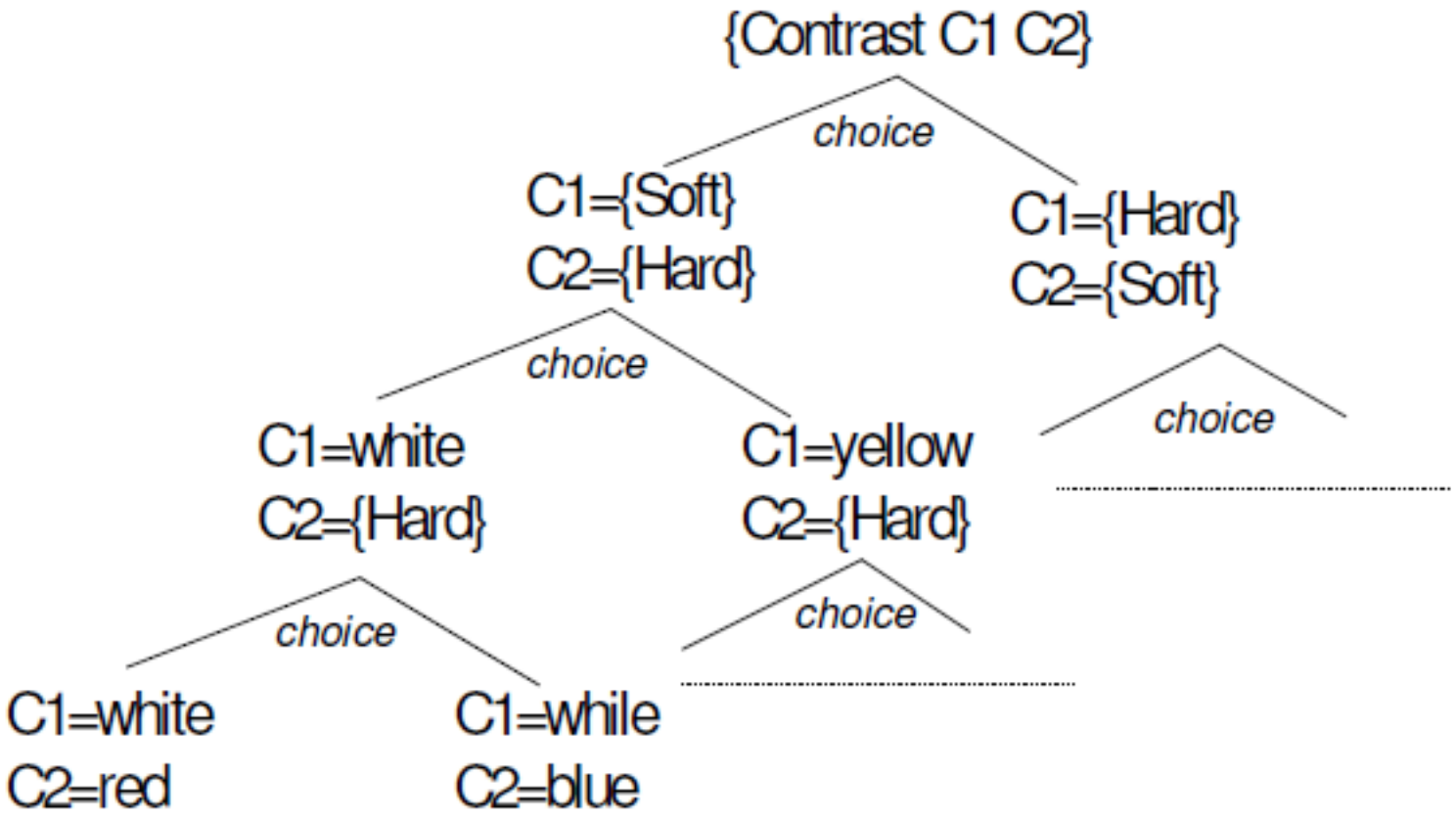
```
fun {Soft} choice white [] yellow end end
fun {Hard} choice red [] blue end end

proc {Contrast C1 C2}
   choice  C1={Soft} C2={Hard}
         [] C1={Hard} C2={Soft} end
end

fun {Suit}
   Shirt Pants Socks
in {Contrast Shirt Pants}
    {Contrast Pants Socks}
    if Shirt==Socks then fail end
      suit(Shirt Pants Socks)
end
```

Search Tree with Choices



`{Solve F}` returns a lazy list of solution for a relational program

Search Tree with Choices




```
{Solve (fun $ C1 C2 in {Contrast C1 C2} p(C1 C2) end)}
```

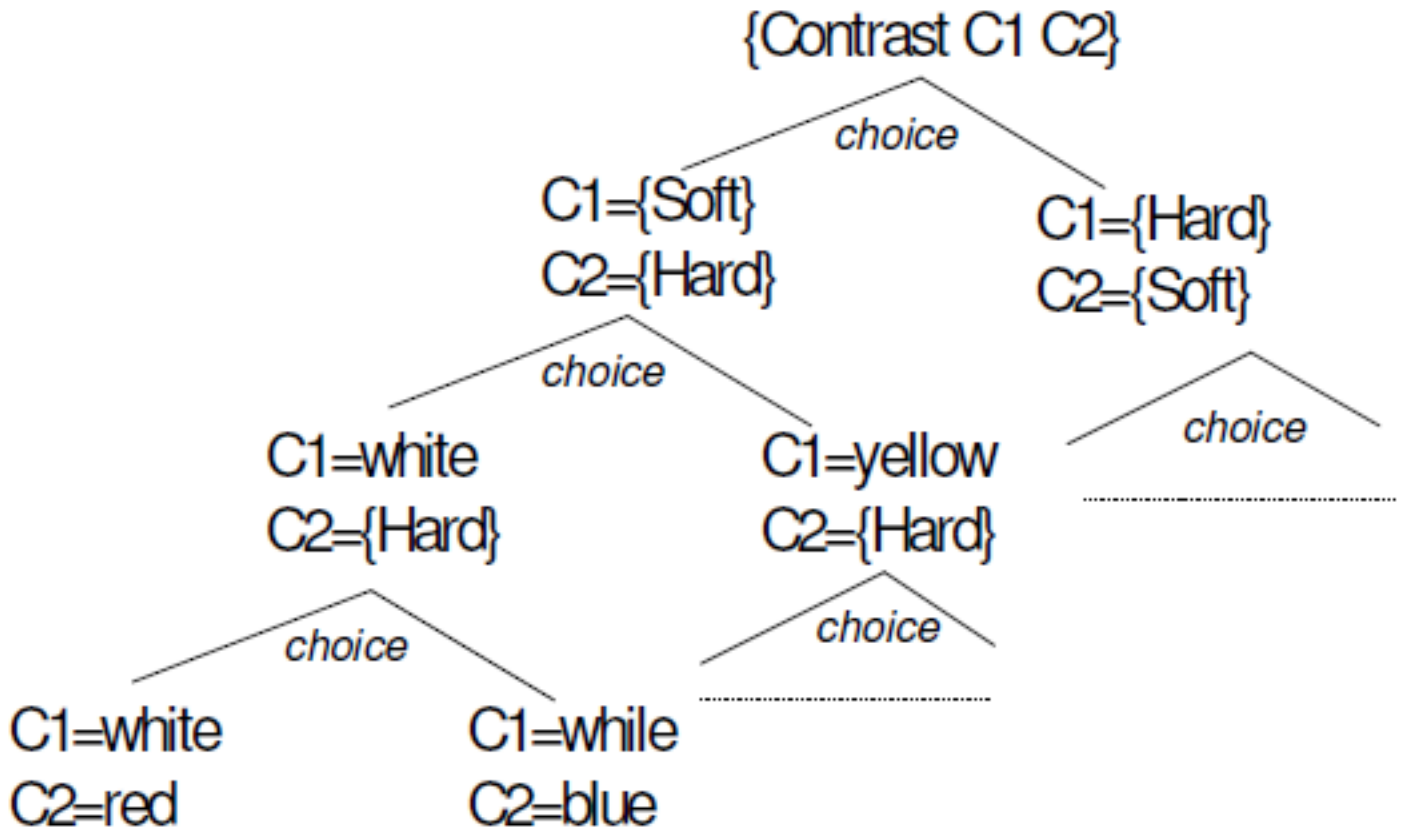
One Solution

- Due to the use of lazy list we can return some or all of the solutions.

- Example:

```
fun {SearchOne F}
   L={Solve F}
in
    if L==nil then nil else [L.1] end
end
```

All Solution




```
{SearchAll (fun $ C1 C2 in {Contrast C1 C2}
                p(C1 C2) end)}
```

Code to Search All Solutions

- Due to lazy list, need to touch every element in the entire list for all solutions.

```
fun {SearchAll F}
   L={Solve F}
   proc {TouchAll L}
      if L==nil then nil
      else {TouchAll L.2} end
   end
in
   {TouchAll L}
   L
end
```

Example

- Note that

```
{Browse {SearchAll Suit}}

[suit(white red yellow)
 suit(while blue yellow)
 suit(yellow red white)
 suit(yellow blue white)
 suit(red white blue)
 suit(red yellow blue)
 suit(blue white red)
 suit(blue yellow red)
]
```

Numeric Example

```
fun {Digit}
   choice 0 [] 1 [] 2 [] ... [] 9 end
end
fun {TwoDigit}
   10*{Digit}+{Digit}
end
fun {StrangeTwoDigit}
   {Digit}+10*{Digit}
end
```

From Functions to Procedures

```
fun {Append A B}
   case A of nil then B
      [] X|As then X|{Append As B} end
end
```

```
proc {Append A B ?C}
   case A of nil then C=B
      [] X|As then Cs in
         C=X|Cs  {Append As B Cs} end
end
```

To Nondeterministic Relations

- Use choice and all parameters may be output.

```
proc {Append ?A ?B ?C}
   choice
         A=nil C=B
      [] As Cs X in
         A=X|As  C=X|Cs  {Append As B Cs}
   end
end
```

Prolog – a logic language

- append relation.

```
append(nil,Ys,Ys.
append([X|Xs],Ys,[X|Zs])
            :- append(Xs,Ys,Zs)
```

- relationships.

```
parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
grandfather(X,Y):-father(X,Z),parent(Z,Y).
son(X,Y):-male(X),parent(Y,X).
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-parent(X,Z),ancestor(Z,Y)
```

Constraint Programming

- Ultimate in declarative modelling.
- Here, we focus on Finite Domain Constraint Programming CP(FD);
- CP initially conceived as framework CLP(X) [Jaffar, Lassez 1987]
- Within CP(FD), we focus on constraint-based tree search

Definitions and Notations

- **Finite domain:**
  - is a finite set of nonnegative integers.
  - The notation `m#n` stands for the finite domain `m...n`.
- **Constraints** over finite domains**:**
  - is a formula of predicate logic.
  - Examples:
    - $X=5$
    - $X \in 1\#9$
    - $X^2-Y^2=Z^2$

Constraint Solving

- **Given**: a satisfiable constraint C and a new constraint C'.
- **Constraint solving** means deciding whether $C \wedge C'$ is satisfiable.
- A finite domain problem has **at most finitely many solutions**, provided we consider only variables that occur in the problem.
- **Example**:

  C: $n > 2$

  C': $a^n + b^n = c^n$

Constraint Solving

- Constraint solving is not possible for general constraints.
- Constraint programming separates constraints into:
  - **basic constraints**: complete constraint solving
  - **non-basic constraints**: propagation-and-search (incomplete)

Basic Constraints in Finite Domain Constraint Programming

- **Basic constraints** are conjunctions of constraints of the form `X ∈ S`, where `S` is a finite set of integers.
- **Constraint solving** is done by intersecting domains.
- Example:

  C = (X∈{1..10} ∧ Y∈{9..20})

  C' = (X∈{9..15} ∧ Y∈{14..30})

- In practice, we keep a ***solved form***, storing the current domain of every variable.

Propagate-and-Search

- **Keep partial information**. During the calculation, we keep partial information about a solution, e.g. `X>10`
- **Use local deduction**. Each of the constraints uses the partial information to deduce more information, e.g. combining `X<Y` and `X>10`, we get `Y>11` (assuming `Y` is an integer).
- **Do controlled search**. When no more local deductions can be done, then we have to search.
  - A **search step** consists in splitting a CSP P into two new problems, $(P \wedge C)$ and $(P \wedge \neg C)$, where C is a new constraint.
  - Since each new problem has an additional constraint, it can do new local deductions.
  - To find the solutions of P, it is enough to take the union of the solutions of the two new problems.

A Complete Constraint Program

- **Problem**: Design a rectangle out of 24 unit squares so that its perimeter is exactly 20?
- **Encoding**: denote x and y the lengths of the rectangle's sides. Then we get two constraints:
  - $x * y = 24$
  - $2 * (x + y) = 20$, or equivalently $x + y = 10$
- **Basic constraints**: $x \in \{1, 2, \ldots, 9\}$ and $y \in \{1, 2, \ldots, 9\}$, because x and y are strict positive integers and $x + y = 10$

Propagators

- The two initial constraints can be viewed as propagators because they can be used to do local deductions:
  - $x * y = 24$
  - $x + y = 10$
- Is there any other possible additional constraint?
- Yes, $x \leq y$
  - It does no harm (since we can always flip a rectangle over)
  - It will make the problem's solution easier (technically, it reduces the size of the search space).

Oz Constraints Programming Notations

- **Basic constraints** are denoted with "`::`" symbol, e.g. the Oz notation `X::1#9` means $x \in \{1, 2, \ldots, 9\}$.

- **Propagators** are denoted by adding the colon : to their name, e.g. `X*Y=:24`, `X+Y=:10`, and `X=<:Y`

- A **computation space** contains the propagators and the basic constraints on the problem variables.
- **Example**: the first computation space is denoted by S_1:
  - `X*Y=:24 X+Y=:10 X=<:Y // X::1#9 Y::1#9`

Local Deductions I

- Each propagator now tries to do local deductions.
- For example, the propagator `X*Y=:24` notices that since `Y` is at most `9`, that `X` cannot be `1` or `2`.
- Therefore `X` is at least `3`. It follows that `Y` is at most `8`.
- The same reasoning can be done with `X` and `Y` reversed.
- So, the propagator updates the computation space:
  - *S*1 : `X*Y=:24 X+Y=:10 X=<:Y // X::3#8 Y::3#8`

Local Deductions II

- Now the propagator `X+Y=:10` enters the picture.
- It notices that since `X` cannot be `2`, therefore `Y` cannot be `8`.
- Similarly, `X` cannot be `8` either.

- This gives
  - *S*1 : `X*Y=:24 X+Y=:10 X=<:Y // X::3#7 Y::3#7`

Local Deductions III

- Now, the propagator `X*Y=:24` can do more deduction.
- Since `X` is at most `7`, therefore `Y` must be at least `4`.
- If `Y` is at least `4`, then `X` must be at most `6`.

- This gives
  - *S*1 : `X*Y=:24 X+Y=:10 X=<:Y // X::4#6 Y::4#6`
- At this point, none of the propagators sees any opportunities for adding information.
- We say that the computation space has become **stable**.

Search I

- How do we continue? We have to make a guess.
- We get two computation spaces: one in which X=4 and another in which X≠4. This gives
  - *S*2 : `X*Y=:24 X+Y=:10 X=<:Y // X=4 Y::4#6`
  - *S*3 : `X*Y=:24 X+Y=:10 X=<:Y // X::5#6 Y::4#6`

Search II

- The local deductions give the computation space S2:
- *S2* : `X*Y=:24 X+Y=:10 X=<:Y // X=4 Y=6`
- At this point, each of the three propagators notices that it is completely solved (it can never add any more information) and therefore removes itself from the computation space. We say that the propagators are **entailed**.
- This gives
  - *S2* : *(empty)* `// X=4 Y=6`
- The result is a solved computation space. It contains the solution `X=4 Y=6`.

Search III

- Local deductions for *S*3:
  - Propagator `X*Y=:24` deduces that `X=6 Y=4` is the only possibility consistent with itself.
  - Then propagator `X=<:Y` sees that there is no possible solution consistent with itself.
  - This causes the space to fail:
  - *S*3 : *(failed)*
  - A failed space has no solution.
- We conclude that the only solution is `X=4 Y=6`.

A Mozart Implementation I

- We define the problem by writing a one argument procedure whose argument is the solution.
- Running the procedure sets up the basic *constraints*, the *propagators*, and selects a *distribution strategy*.
- The **distribution strategy** defines the "guess" that splits the search in two.

A Mozart Implementation II

```
declare
proc {Rectangle ?Sol}
   sol(X Y)=Sol
in
   X::1#9 Y::1#9
   X*Y=:24 X+Y=:10 X=<:Y
   {FD.distribute naive Sol}
end
{Browse {SearchAll Rectangle}}
```

A Mozart Implementation III

- The **solution** is returned as the tuple `sol(X Y)`
- Basic constraints: `X::1#9` and `Y::1#9`
- Propagators: `X*Y=:24`, `X+Y=:10`, and `X=<:Y`
- The `FD.distribute` call selects the distribution strategy.
- The chosen strategy (`naive`) selects the first nondetermined variable in `Sol`, and picks the leftmost element in the domain as a guess.

Summary so far: Computation Space

- Is the fundamental concept used to implement propagate-and-search, which contains basic constraints and propagators
- Solving a problem alternates two phases:
  - Local deductions (using the propagators).
  - Search step (when the space is stable). Two copies of the space are first made. A basic constraint *C* is then "guessed" according to the distribution strategy. The constraint *C* is then added to the first copy and ¬*C* is added to the second copy. We then continue with each copy. The process is continued until all spaces are either solved or failed.
- This gives us all solutions to the problem.

Constraint Programming in a Nutshell

SEND

MORE

MONEY

Constraint Programming in a Nutshell

SEND + MORE = MONEY

SEND + MORE = MONEY

Assign distinct digits to the letters

S, E, N, D, M, O, R, Y

such that

```
          S E N D
      +   M O R E
    ---------------
    = M O N E Y
```

holds.

SEND + MORE = MONEY

Assign distinct digits to the letters

S, E, N, D, M, O, R, Y

such that

```
  S E N D
+ M O R E
---------
= M O N E Y
```

holds.

Solution

```
  9 5 6 7
+ 1 0 8 5
---------
= 1 0 6 5 2
```

Modeling

- Formalize the problem as a ***constraint problem***:

- number of variables: $\mathbf{n}$
- constraints: $\mathbf{c}_1, \ldots, \mathbf{c}_m \subseteq Z^n$
- problem: Find $\mathbf{a} = (\mathbf{v}_1, \ldots, \mathbf{v}_n) \in Z^n$ such that $\mathbf{a} \in \mathbf{c}_i$, for all $\mathbf{1} \leq \mathbf{i} \leq \mathbf{m}$

A Model for **MONEY**

- number of variables: **8**
- constraints:

$$c_1 = \{ (S,E,N,D,M,O,R,Y) \in Z^8 \mid 0 \le S,\ldots,Y \le 9 \}$$

$$c_2 = \{ (S,E,N,D,M,O,R,Y) \in Z^8 \mid$$

$$1000*S + 100*E + 10*N + D$$

$$+ 1000*M + 100*O + 10*R + E$$

$$= 10000*M + 1000*O + 100*N + 10*E + Y\}$$

A Model for **MONEY** (continued)

- more constraints

$c_3 = \{ (S,E,N,D,M,O,R,Y) \in Z^8 \mid S \neq 0 \}$

$c_4 = \{ (S,E,N,D,M,O,R,Y) \in Z^8 \mid M \neq 0 \}$

$c_5 = \{ (S,E,N,D,M,O,R,Y) \in Z^8 \mid S \ldots Y$ **all different** $\}$

Solution for **MONEY**

$$c_1 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid 0 \le S,\ldots,Y \le 9\}$$

$$c_2 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid 1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E = 10000*M + 1000*O + 100*N + 10*E + Y\}$$

$$c_3 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid S \neq 0\}$$

$$c_4 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid M \neq 0\}$$

$$c_5 = \{(S,E,N,D,M,O,R,Y) \in Z^8 \mid S\ldots Y \text{ all different}\}$$

- **Solution:** $(9,5,6,7,1,0,8,2) \in Z^8$

A Mozart Implementation

```
proc {Money Root}
   S E N D M O R Y in
   Root=sol(s:S e:E n:N d:D m:M o:O r:R y:Y)
   Root ::: 0#9
   {FD.distinct Root}
   S \=: 0   M \=: 0
                1000*S + 100*E + 10*N + D
   +            1000*M + 100*O + 10*R + E
   =: 10000*M + 1000*O + 100*N + 10*E + Y
   {FD.distribute ff Root}
end

{Browse {SearchAll Money}}
```

Modeling

Propagation

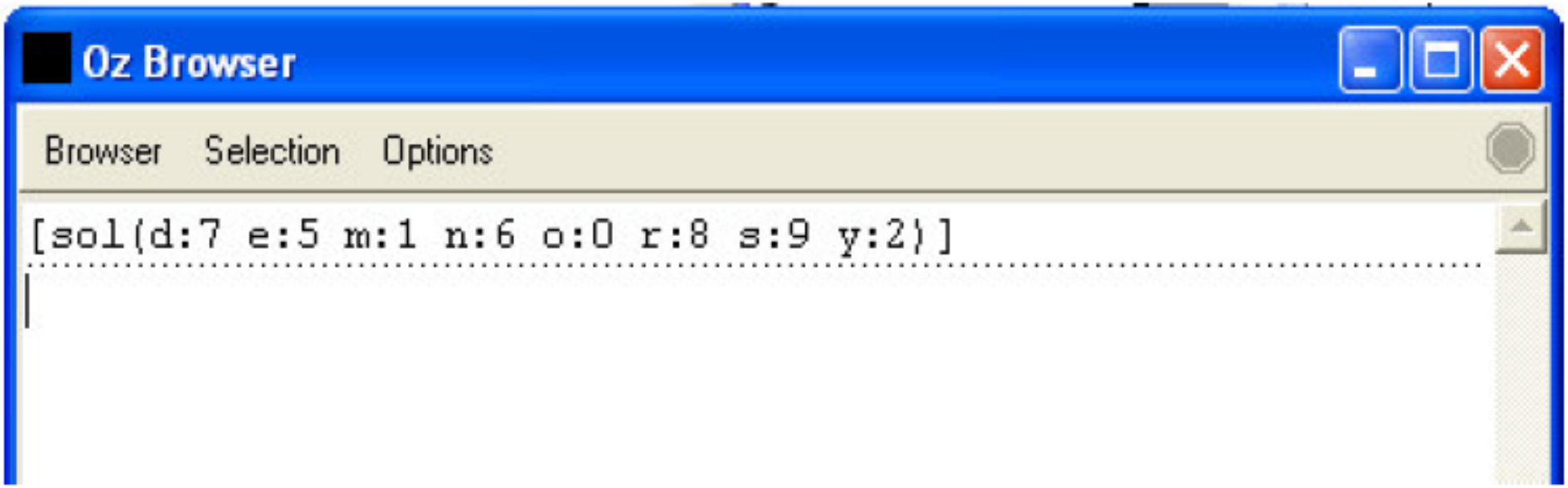
Branching

Exploration

Money Demo

- To display the solutions:

```
{Browse {SearchAll Money}}
```



Relation to Integer Programming

- More general notion of problem; constraints can be any relation, not just arithmetic or even just linear arithmetic constraints
- De-emphasize optimization (optimization as after-thought)
- Focus on software engineering
  - no push-button solver, but glass-box or no-box
  - experimentation platforms
  - extensive support for “performance debugging”

Constraint Programming Systems

- Role: support elements of constraint programming
  - Provide propagation algorithms for constraints
    - all different (e.g. wait for fixing)
    - summation (e.g. interval consistency)
  - Allow choice of branching algorithm (e.g. first-fail)
  - Allow choice of exploration algorithm (e.g. depth-first search)

Programming Systems for Finite Domain Constraint Programming

- Finite domain constraint programming libraries
  - PECOS [Puget 1992]
  - ILOG Solver [Puget 1993]
- Finite domain constraint programming languages
  - CHIP [Dincbas, Hentenryck, Simonis, Aggoun 1988]
  - SICStus Prolog [Haridi, Carlson 1995]
  - Oz [Smolka and others 1995]
  - CLAIRE [Caseau, Laburthe 1996]
  - OPL [van Hentenryck 1998]

Issues in Propagation

- **Expressivity**: What kind of information can be expressed as propagators?
- **Completeness**: What behavior can be expected from propagation?
- **Efficiency**: How much computational resources does propagation consume?

Summary

- Relational Programming
- Choice and Fail Operations
- Constraint Programming in a Nutshell
- Elements of Constraint Programming
- Constraint Programming in Oz
- Constraint Programming Techniques