Constraint Programming with COMET

Laurent Michel ldm@engr.uconn.edu
Pascal Van Hentenryck pvh@cs.brown.edu

Overview

- The COMET Platform
- Core Language
- The CP Solver
  - Declarative Model
  - Search Procedures
- Demo

COMET

- An optimization platform
  - Constraint-based Local Search (CBLS)
  - Constraint Programming (CP)
  - Mathematical programming (MP)
- Availability

Windows	32
MacOS	32/64
Linux	32/64

Integrating Code with COMET

- Options available
  - Extend COMET in COMET
    - User defined constraints (in CBLS and FD)
  - Extend COMET in C++
    - Call your C++ code from COMET. Plugin architecture.
  - *Embed* COMET in C++
    - Call COMET from C++

Integrating Data Sources with COMET

- Database connectivity
  - ODBC 2.0 (on all platforms)
- Data files
  - XML reading/writing

User Interface with COMET

- Version 1.2 (and earlier)
  - Cocoa visualization on MacOS
  - Gtk visualization on Linux
  - Nothing on windows

- Version 1.3 (or 2.0...)
  - QT-based visualization
  - On all platforms!

Writing COMET programs?

- On version 1.2
  - Development Studio on MacOS
  - Emacs + command line on Linux
  - Emacs + command line on Windows
- On version 1.3 (or 2.0...)
  - Development Studio with QT on all platforms

Debugging COMET programs?

- On version 1.2
  - Alpha version of a GUI debugger on Linux (GTK)
  - Alpha version of a GUI debugger on MacOS (Cocoa)
  - Alpha version of a text debugger on windows
- On version 1.3 (or 2.0...)
  - GUI debugger on all platforms (QT again!)

Modeling with COMET

- Modeling power
  - High level models for CBLS and CP
  - rich language of constraints and objectives
  - vertical extensions

Solving with COMET

- Search
  - a unique search language for CBLS, CP, MP
- Hybridization
  - Solvers are first-class objects

Hybrids 1

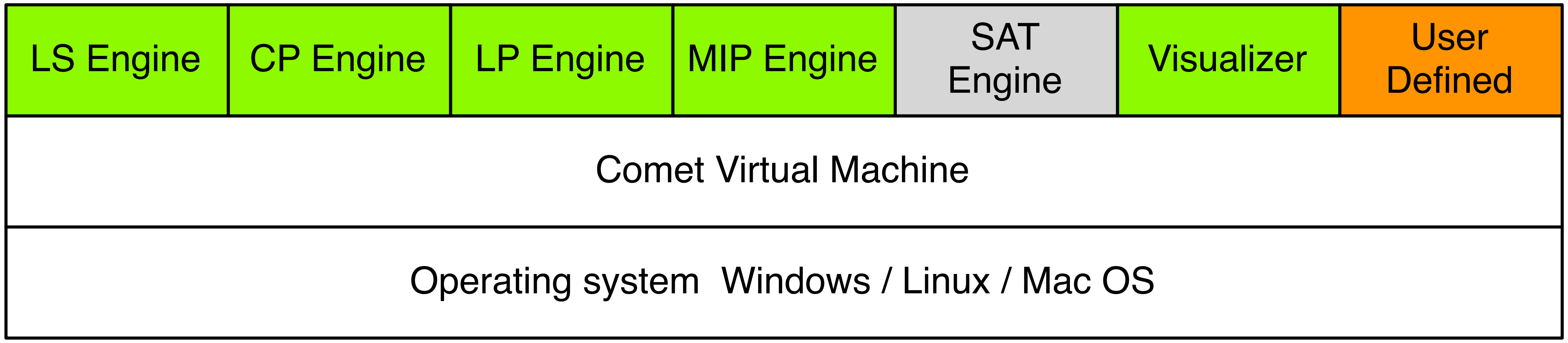
- Two LP/MIP Solvers
  - lpsolve
  - coin-Clp
- Techniques supported through model composition
  - Model chaining
  - Column generation
  - Benders decomposition

Hybrids 2

- Combine CP + LS
  - LS for high-quality solutions quickly (and speed up the CP proof)
  - CP for optimality proof - completeness
- Composition?
  - Sequential
  - Parallel
- Communication?
  - Bounds
  - Actual solution, frequencies, ...

Architecture

Loadable plugins



LS Engine

CP Engine

LP Engine

MIP Engine

SAT Engine

Visualizer

User Defined

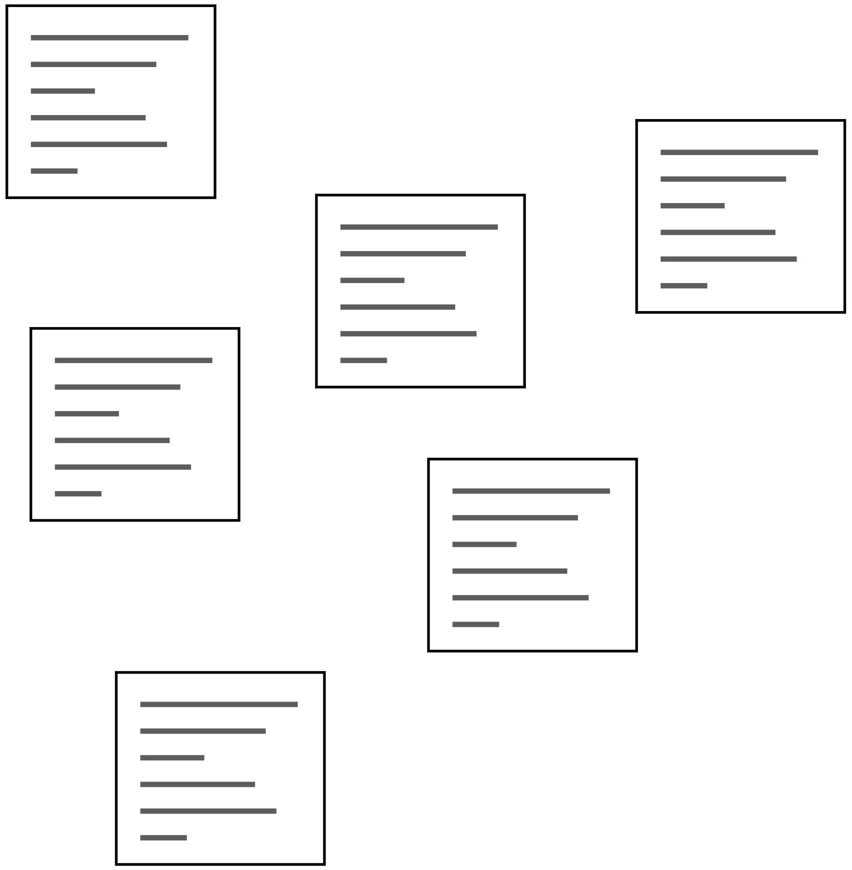
Comet Virtual Machine

Operating system Windows / Linux / Mac OS

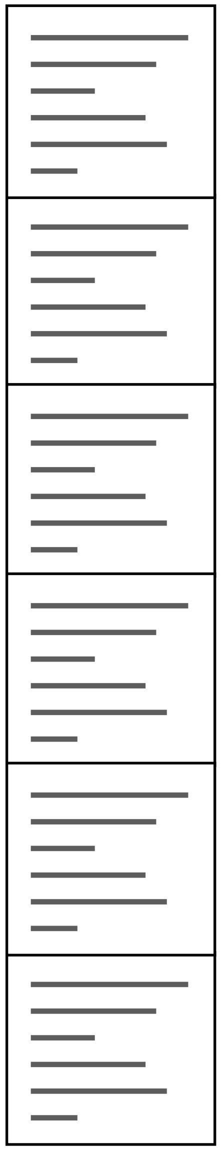
Core Language

- Similar to C++ or Java
  - Statically typed
  - Strongly typed
- Abstractions
  - Classes
  - Interfaces
- Control
  - All the usual gizmos
  - Additional looping / branching construction

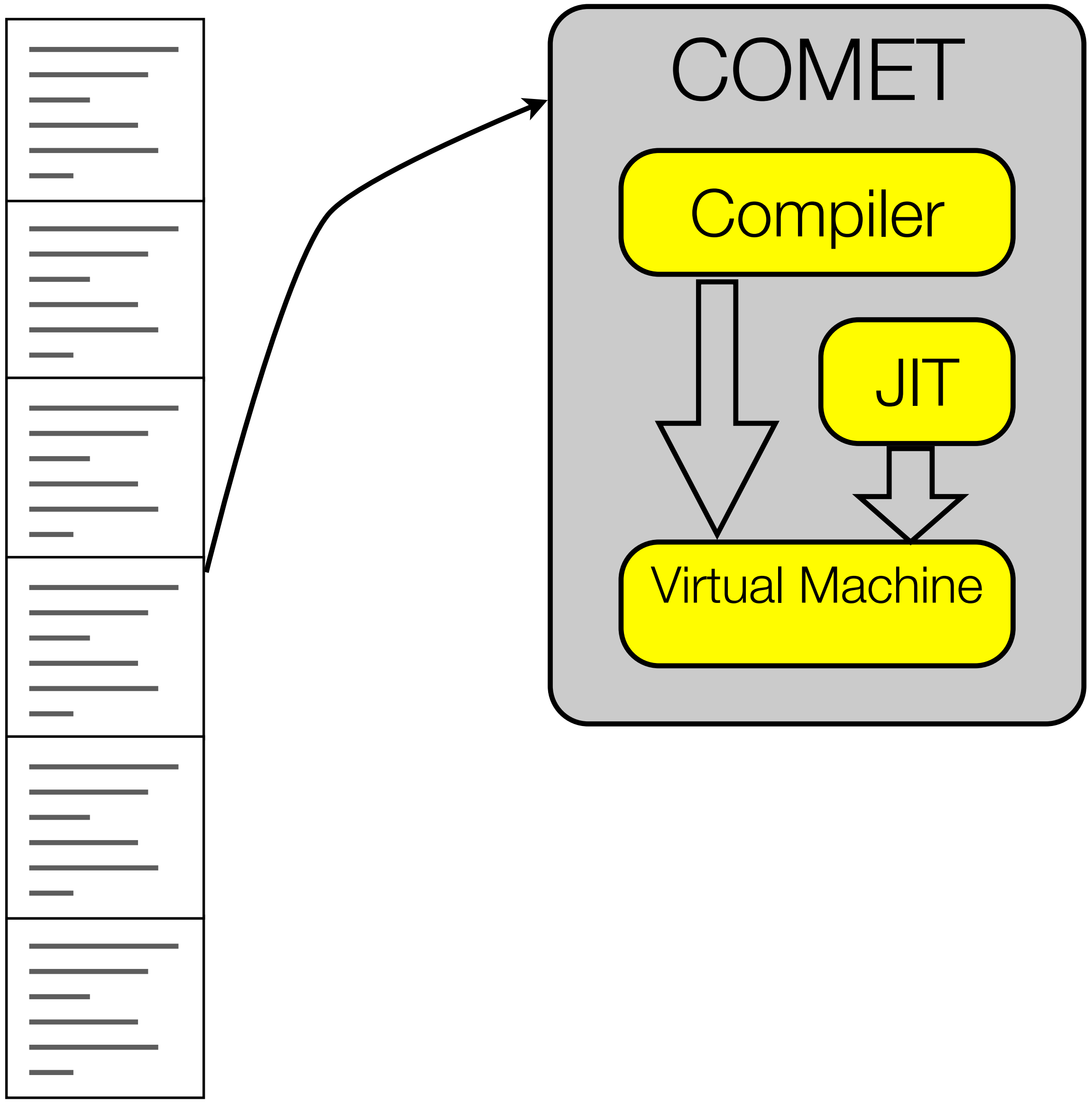
Workflow



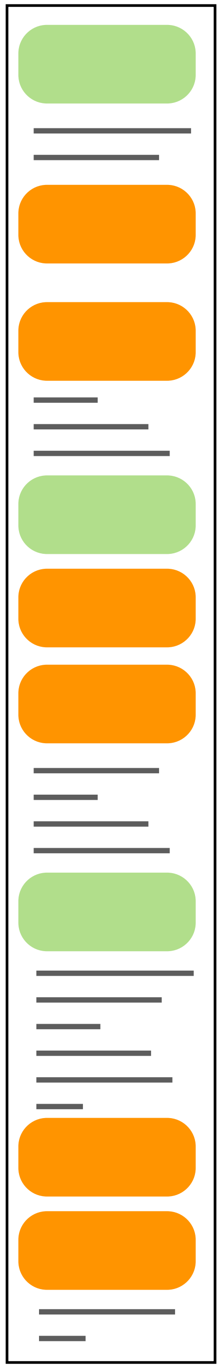
Workflow



Workflow



Source Organization

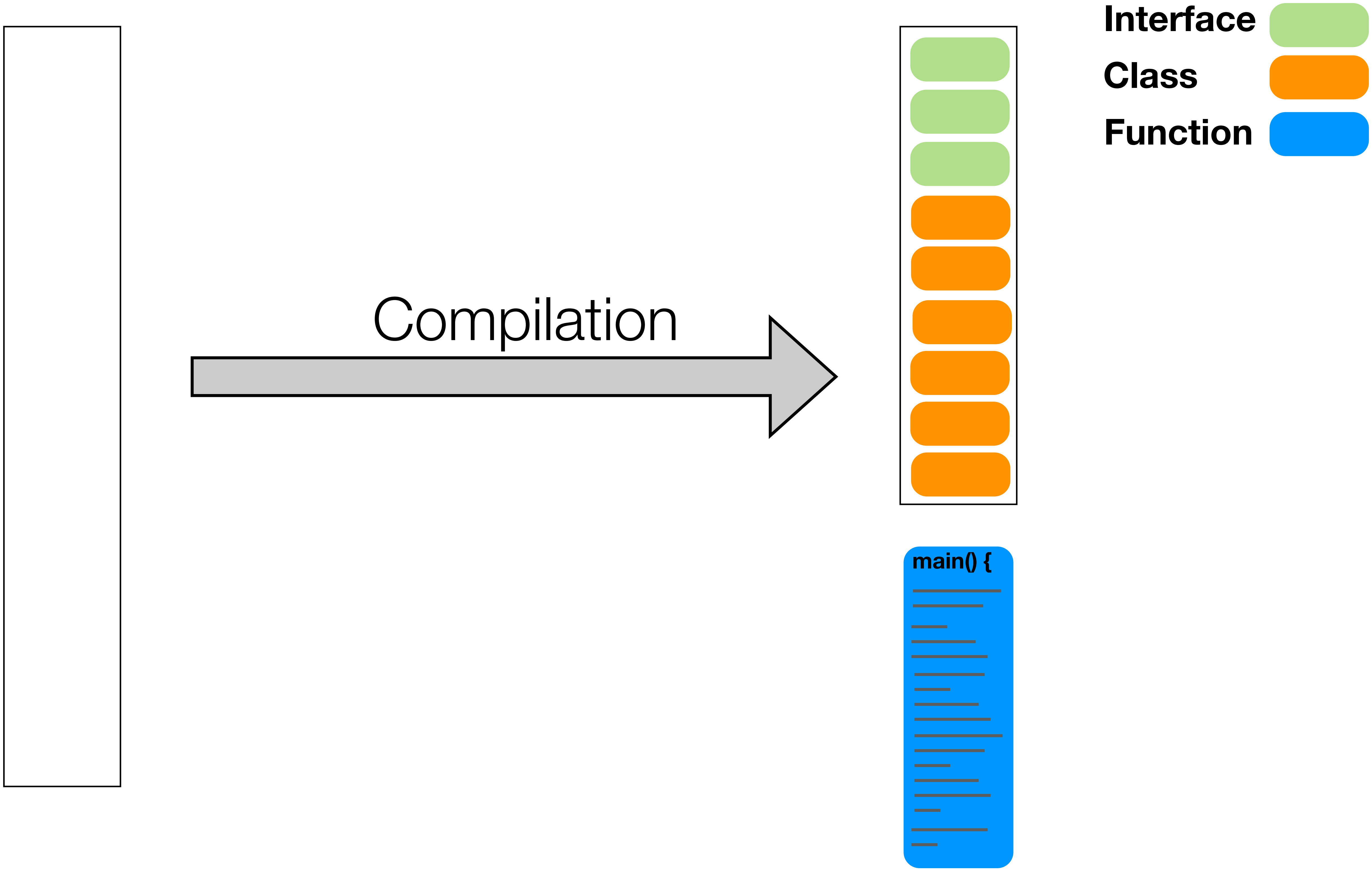


Interface 🟩

Class 🟧

Function 🟦

Source Organization

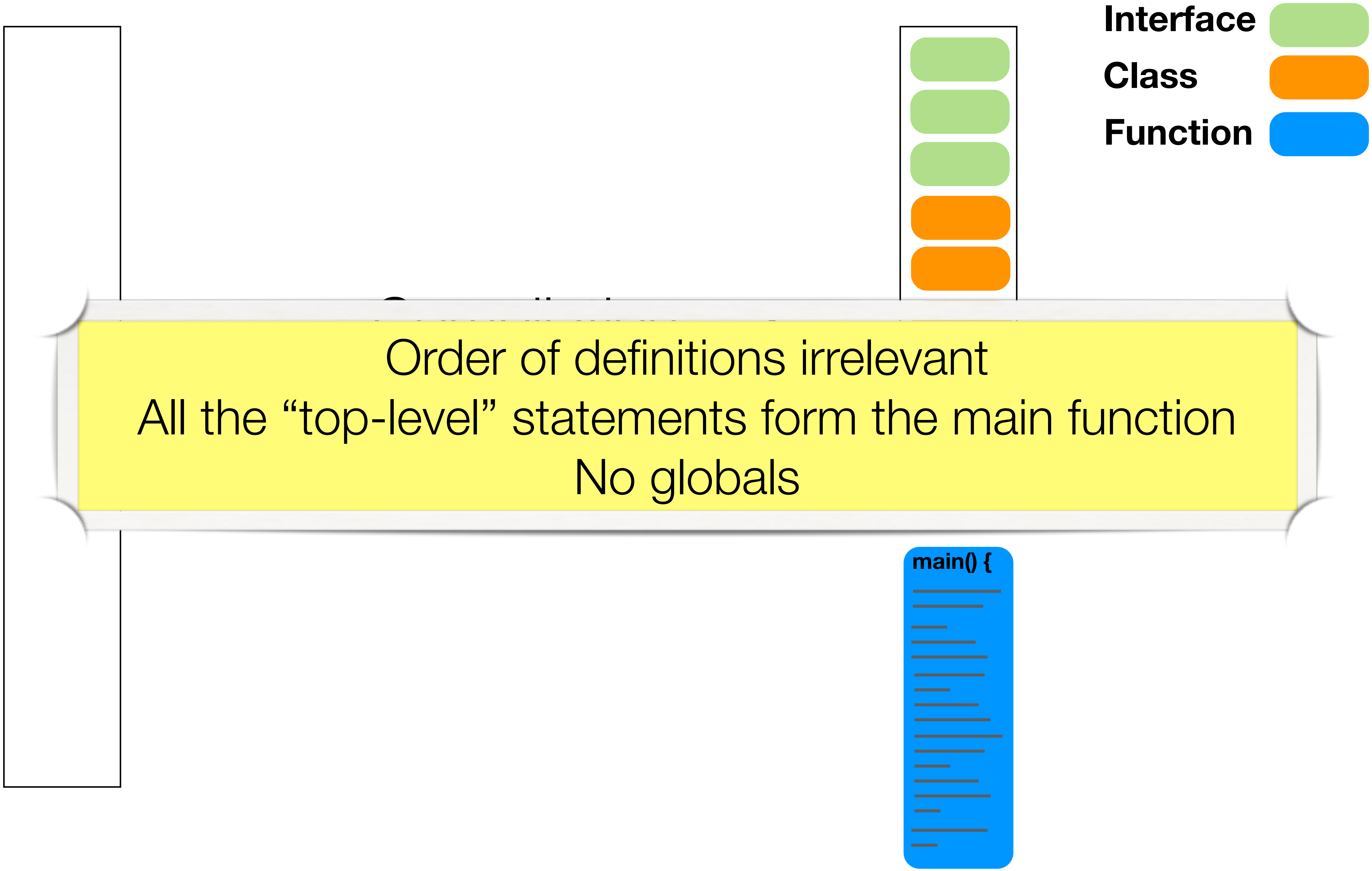


Interface 🟩

Class 🟧

Function 🟦

Source Organization



- **Interface** 🟩
- **Class** 🟧
- **Function** 🟦

Order of definitions irrelevant

All the "top-level" statements form the main function

No globals

Basic Language support

- You can define
  - Classes
  - Functions
  - Interfaces
- All the traditional C++/Java - like statements
- Parameter passing is by value
  - Integer, Float,Boolean classes like in Java
- IO
  - stream-based (cin/cout) like in C++

Data support

- Data support
  - array, matrices, sets, stack, queues, dictionaries
- Expressions
  - Rich expression language with aggregates for arithmetic and sets

```
int x = sum(i in R) x[i];
int y = prod(i in R) x[i];
set{int} a = setof(i in R) (x[i]i%2==0);
set{int} b = collect(i in R) x[i];
```

  - Slicing

```
int mx[i in 1..10,j in 1..10] = i * 10 + j;
int []col3 = all(i in 1..10) mx[i,3];
int []row4 = all(i in 1..10) mx[4,i];
int []diag = all(i in 1..10) mx[i,i];
```

Extra Control: Forall Loops

- Basic
- With ordering

```
forall(i in S)
  BLOCK

forall(i in S : p(i))
  BLOCK

forall(i in S : p(i)) by (f(i))
  BLOCK
```

Extra Control: Branching - Selectors

- Randomized, Minimum, Maximum
- Semi-greedy

```
select(i in S)
  BLOCK

selectMin(i in S)(f(i))
  BLOCK

selectMax(i in S)(f(i))
  BLOCK

selectMin[k](i in S)(f(i))
  BLOCK

selectMax[k](i in S)(f(i))
  BLOCK
```

```
select(i in S : p(i))
  BLOCK

selectMin(i in S : p(i))(f(i))
  BLOCK

selectMax(i in S : p(i))(f(i))
  BLOCK

selectMin[k](i in S : p(i))(f(i))
  BLOCK

selectMax[k](i in S : p(i))(f(i))
  BLOCK
```

Extra Control: Branching - Selectors

- Randomized, Minimum, Maximum
- Semi-greedy

```
select(i in S)
  BLOCK

selectMin(i in S)(f(i))
  BL
  
sel
  BLOCK

selectMin[k](i in S)(f(i))
  BLOCK

selectMax[k](i in S)(f(i))
  BLOCK
```

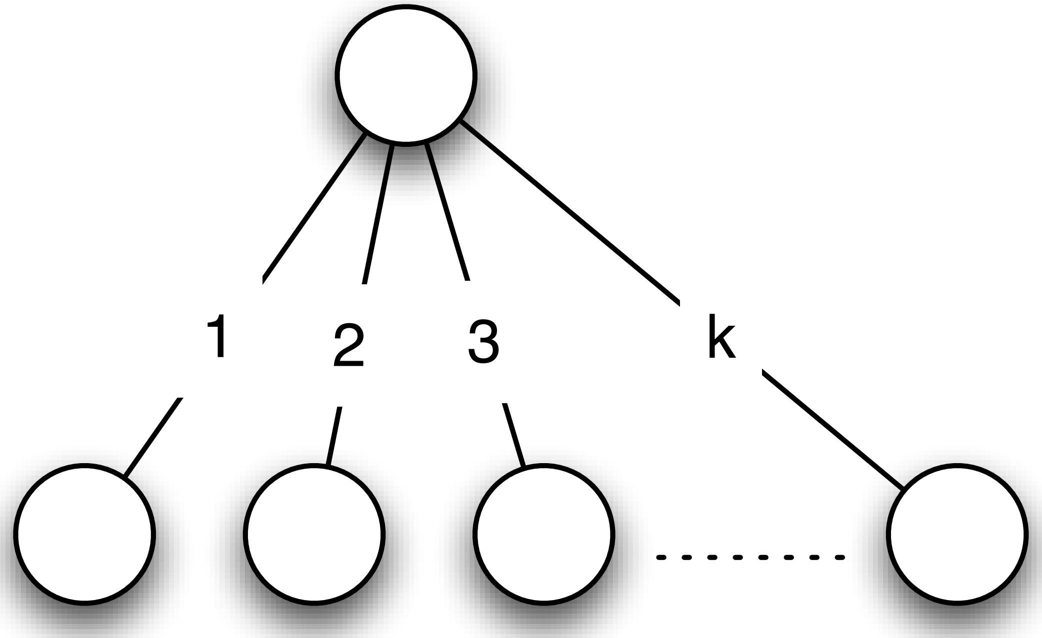
```
select(i in S : p(i))
  BLOCK

selectMin(i in S : p(i))(f(i))


  BLOCK

selectMin[k](i in S : p(i))(f(i))
  BLOCK

selectMax[k](i in S : p(i))(f(i))
  BLOCK
```

Tie-break Broken uniformly at random
Semi-greedy Selectors respect probability distributions

Extra Control: Non-determinism

- Let us express choices
  - Binary

```
try<c>
        BLOCK1
   |    BLOCK2
   |    BLOCK3
   ...
   |    BLOCKK
```

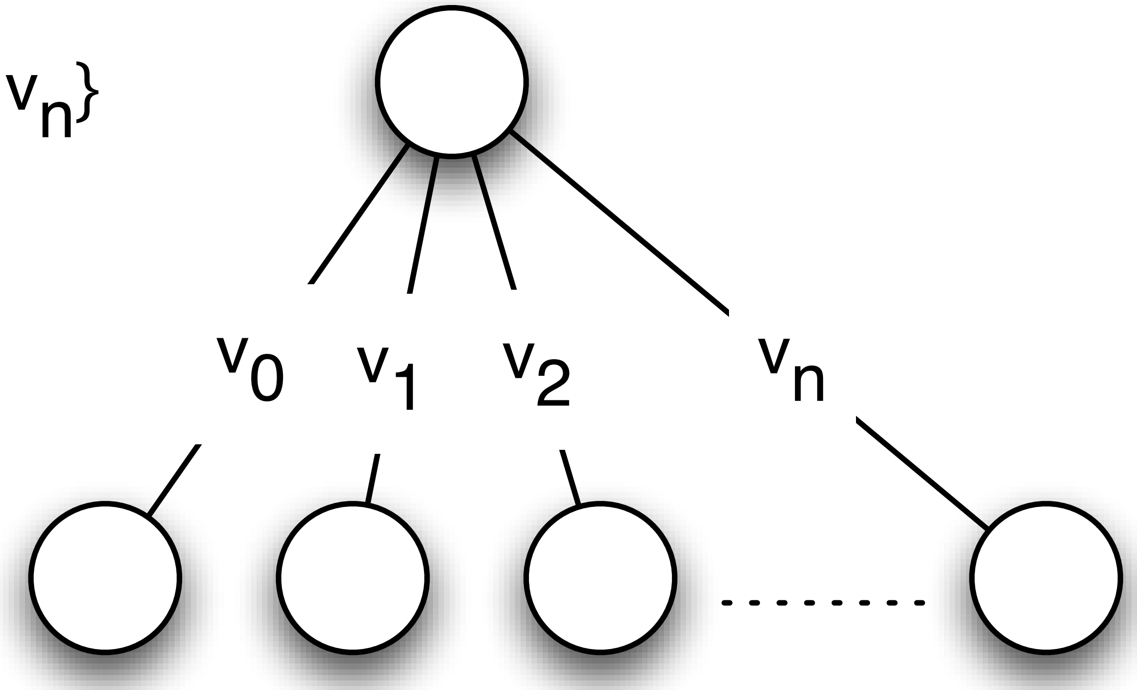


Extra Control: Non-determinism

- Let us express choices
  - N-ary
  - Branches given by set S

```
tryall<c>(i in S)
   BLOCK
```

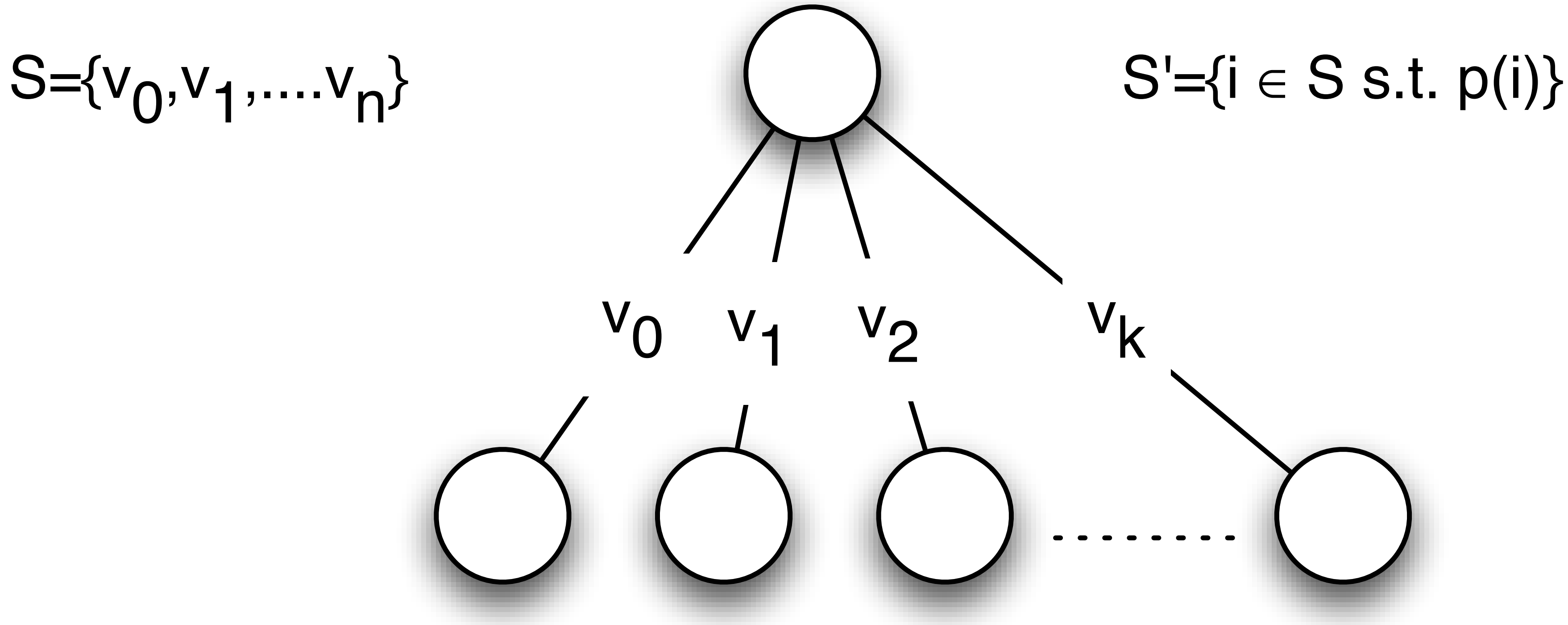
$S=\{v_0,v_1,....v_n\}$



Extra Control: Non-determinism

- Let us express choices
  - N-ary
  - Branches given by subset of S satisfying p(i)

```
tryall<c>(i in S : p(i))
   BLOCK
```



$S=\{v_0,v_1,....v_n\}$

$S'=\{i \in S \text{ s.t. } p(i)\}$

Extra Control: Non-determinism

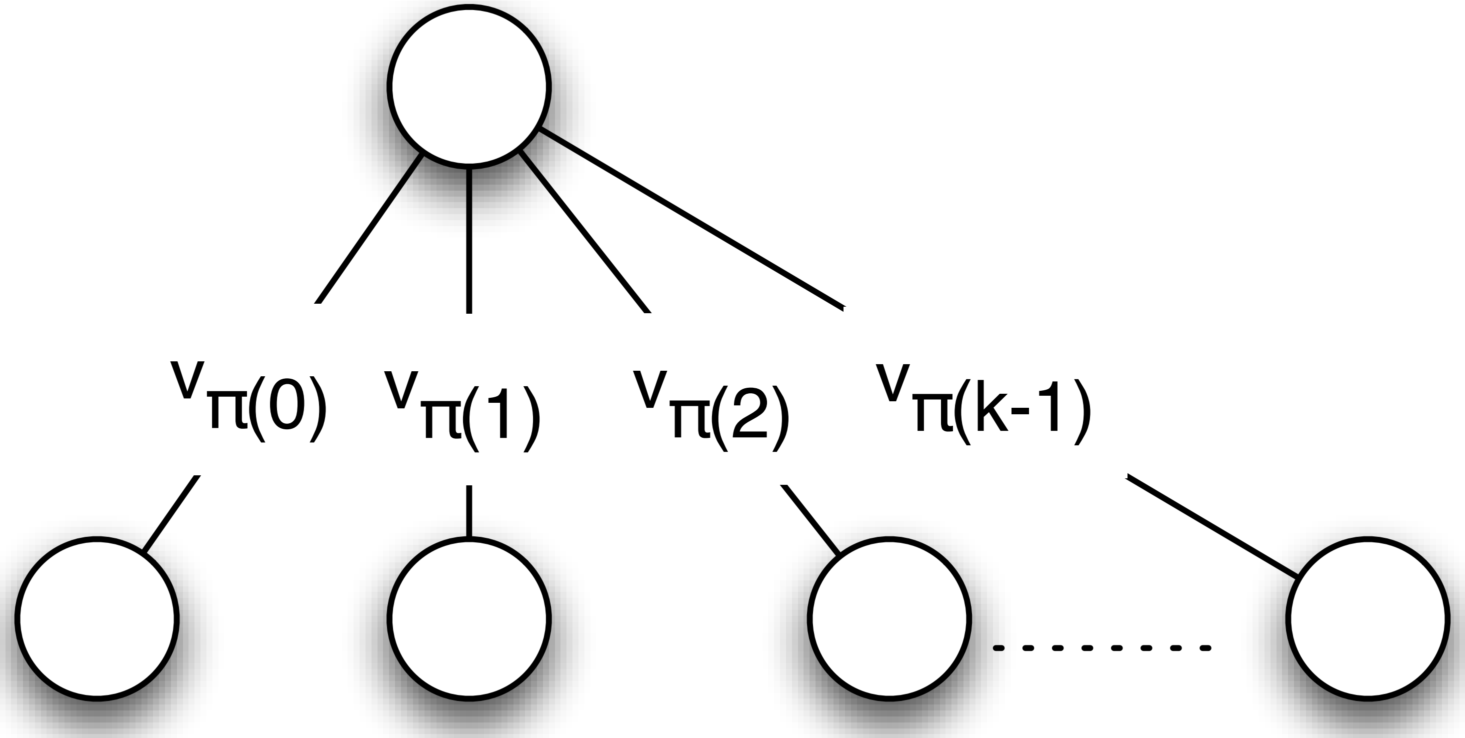
- Let us express choices
  - N-ary
  - Consider choices in order of increasing f(i)

```
 tryall<c>(i in S : p(i)) by (f(i))
    BLOCK
```

$S=\{v_0,v_1,....v_n\}$

$S'=\{i \in S \text{ s.t. } p(i)\}$, $|S| = k$

π =permutation(0..k-1)
s.t. $i \leq j \Rightarrow f(\pi(i)) \leq f(\pi(j))$



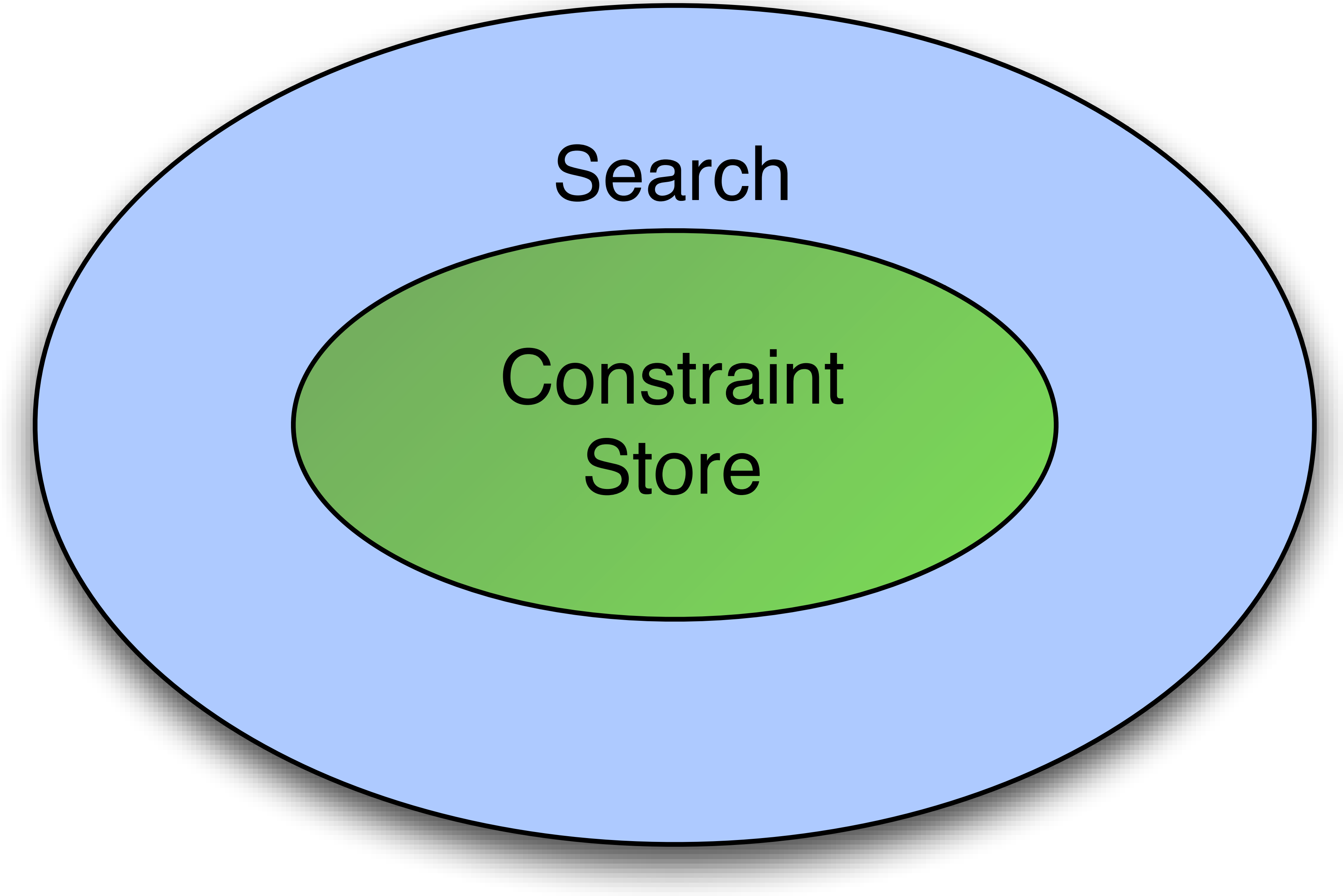
Extra Control: Non-determinism

- Let us express choices
  - N-ary

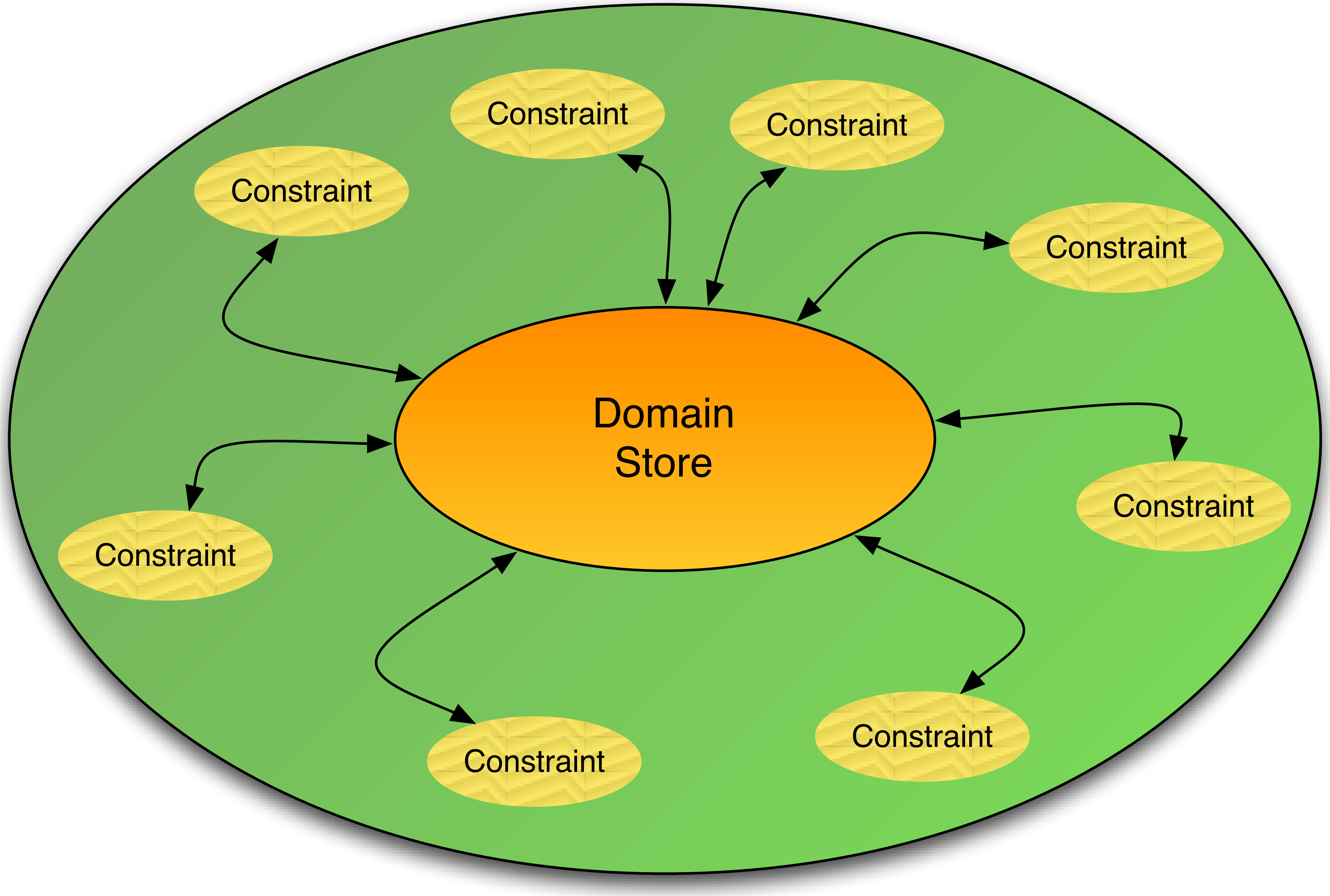
```
tryall<c>(i in S : p(i)) by (f(i))
   BLOCK
onFailure BLOCK2
```

- Adds ability to
  - Execute BLOCK2 when there is a failure
  - Before trying the next choice....

CP Computational Model

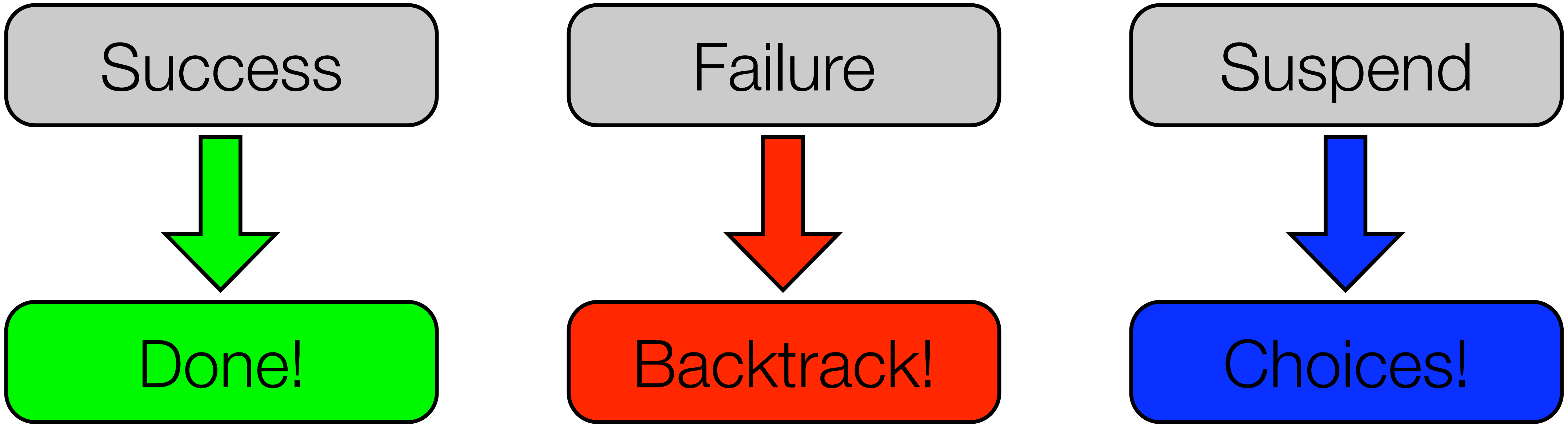


Computational Model



Operationally

- Compute a fixpoint of the constraint set
  - Reason on each constraint C *locally*
    - For every variable X appearing in C: prune D(x)
    - *Propagate* the impact to other constraints using X
  - Stop when no more changes
- Outcomes ?



Success → Done!

Failure → Backtrack!

Suspend → Choices!

Solvers

- Computational Model embedded in a solver
- Comet supports several solvers

```
import cotfd;
Solver<CP>   cp();
```

```
import cotln;
Solver<LP>   lp();
Solver<MIP>  ip();
```

```
import cotls;
Solver<LS>   ls();
```

Importing =
Loading a shared library +
defining all the interfaces +
defining all the classes

Solvers

- Computational Model embedded in a solver
- Comet supports several solvers

```
import cotfd;
Solver<CP>   cp();
```

```
import cotln;
Solver<LP>   lp();
Solver<MIP>  ip();
```

```
import cotls;
Solver<LS>   ls();
```

Importing =
Loading a shared library +
defining all the interfaces +
defining all the classes

Variables

- Variables are declared for a specific Solver
- For finite domain
  - Domain can be a range or a set.

```
import cotfd;
Solver<CP>   cp();


var<CP>{int}       x(cp,D);
var<CP>{bool}      y(cp);

var<CP>{set{int}}  z(cp,1..10);    // In upcoming v1.3
```

Declarative Model

- Model states
  - The nature of the problem
    - Constraint Satisfaction Problem
      - Find ***one*** solution
      - Find ***all*** solution
    - Constraint Optimization Problem
      - Find one global solution.
      - Prove optimality
  - the constraints
    - Arithmetic / Logical / Combinatorial

CSP vs. COP

CSP

```
Solver<CP> m();
...
solve<m> {

   ...
} [using BLOCK]


Solver<CP> m();
...
solveall<m> {

   ...
} [using BLOCK]
```

COP

```
Solver<CP> m();
...
minimize<m> obj
subject to {
   ...
} [using BLOCK]


Solver<CP> m();
...
maximize<m> obj
subject to {
   ...
} [using BLOCK]
```

Stating Constraints

- Constraints should be stated *directly* or *indirectly* via one of...
  - The “solve” block
  - The “subject to” block
  - The “using” block

```
solve<m> {
   m.post(constraint);
}
```

Auto,onDomains,
onBounds, onValues

- Rationale...
  - Constraints can *fail* (prove infeasibility)
  - Constraints posted inside the block trigger backtracking
  - Constraints posted outside these block simply fail
    - [you must check the status manually]

Stating Constraints

- Constraints should be stated *directly* or *indirectly* via one of...
  - The “solve” block
  - The “subject to” block
  - The “using” block

```
solve<m> {
   m.post(constraint,onDomains);
}
```

Auto,onDomains,
onBounds, onValues

- Rationale...
  - Constraints can *fail* (prove infeasibility)
  - Constraints posted inside the block trigger backtracking
  - Constraints posted outside these block simply fail
    - [you must check the status manually]

Arithmetic Constraints

- Use all the traditional arithmetic operators
  - Binary operators: + - * / ^ min max
  - absolute value: abs()
- Use all the relational operators
  - < <= > >= == !=

Element Constraints

- Array and matrix indexing
  - All combinations are allowed
    - Index an array of constants with a variable [ELEMENT]
    - Index a matrix of constants with variable(s) [Matrix ELEMENT]
    - Index an array of variables with a variable
    - Index a matrix of variables with variables(s)

Logical Constraints

- Negation
  - With the ! operator

```
m.post(!b);
```

- Conjunction
  - With the && operator

```
m.post((a < b) && (a < d));
```

- Disjunction
  - With the || operator

```
m.post((a < b) || (a < d));
```

- Implication
  - With the => operator

```
m.post(a => b);
```

Combinatorial Constraints

- The “global” constraints
  - alldifferent
  - cardinalities (at least, at most, exactly)
  - binaryKnapsack, multiKnapsack,binPacking
  - spread, deviation
  - circuit
  - inverse
  - lexleq
  - table
  - sequence
  - scheduling constraints...

First Simple Example

- SEND + MORE = MONEY

```
import cotfd;

Solver<CP> m();
range Digits = 0..9;

var<CP>{int} x[1..8](m,Digits);
var<CP>{int} S = x[1];
var<CP>{int} E = x[2];
var<CP>{int} N = x[3];
var<CP>{int} D = x[4];
var<CP>{int} M = x[5];
var<CP>{int} O = x[6];
var<CP>{int} R = x[7];
var<CP>{int} Y = x[8];
```

First Simple Example

- SEND + MORE = MONEY

```
import cotfd;

Solver<CP> m();
range Digits = 0..9;

var<CP>{int} x[1..8](m,Digits);
var<CP>{int} S = x[1];
var<CP>{int} E = x[2];
var<CP>{int} N = x[3];
var<CP>{int} D = x[4];
var<CP>{int} M = x[5];
var<CP>{int} O = x[6];
var<CP>{int} R = x[7];
var<CP>{int} Y = x[8];
```

```
solve<m> {
   m.post(alldifferent(x));
   m.post(M != 0);
   m.post(S != 0);
   m.post(            1000 * S + 100 * E + 10 * N + D +
                      1000 * M + 100 * O + 10 * R + E ==
          10000 * M + 1000 * O + 100 * N + 10 * E + Y);
}

cout << x << endl;
```

First Simple Example

- SEND + MORE = MONEY

```
import cotfd;

Solver<CP> m();
range Digits = 0..9;

var<CP>{int} x[1..8](m,Digits);
var<CP>{int} S = x[1];
var<CP>{int} E = x[2];
var<CP>{int} N = x[3];
var<CP>{int} D = x[4];
var<CP>{int} M = x[5];
var<CP>{int} O = x[6];
var<CP>{int} R = x[7];
var<CP>{int} Y = x[8];
```

Notes

1. Solve block
2. Default Search
3. Arithmetic constraint
4. One Combinatorial constraint

```
solve<m> {
   m.post(alldifferent(x));
   m.post(M != 0);
   m.post(S != 0);
   m.post(            1000 * S + 100 * E + 10 * N + D +
                      1000 * M + 100 * O + 10 * R + E ==
          10000 * M + 1000 * O + 100 * N + 10 * E + Y);
}

cout << x << endl;
```

Example

- Magic series
  - A serie of length 5

```
  0 1 2 3 4
s[2,1,2,0,0]
```

- Reification (a.k.a. meta-constraint): constraint on constraints

```
import cotfd;
Solver<CP> m();
int n = 20;
range D = 0..n-1;
var<CP>{int} s[D](m,D);
solve<m> {
  forall(k in D)
    m.post(s[k] == sum(i in D) (s[i]==k));

}

cout << s << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail    = " << m.getNFail() << endl;
```

Improving the model : Redundant Constraints

- Add redundant constraint(s)!

$$\sum_{k \in 0..n-1} s[k] = n \qquad \sum_{k \in 0..n-1} k \cdot s[k] = n \qquad \sum_{k \in 0..n-1} (k-1) \cdot s[k] = 0$$

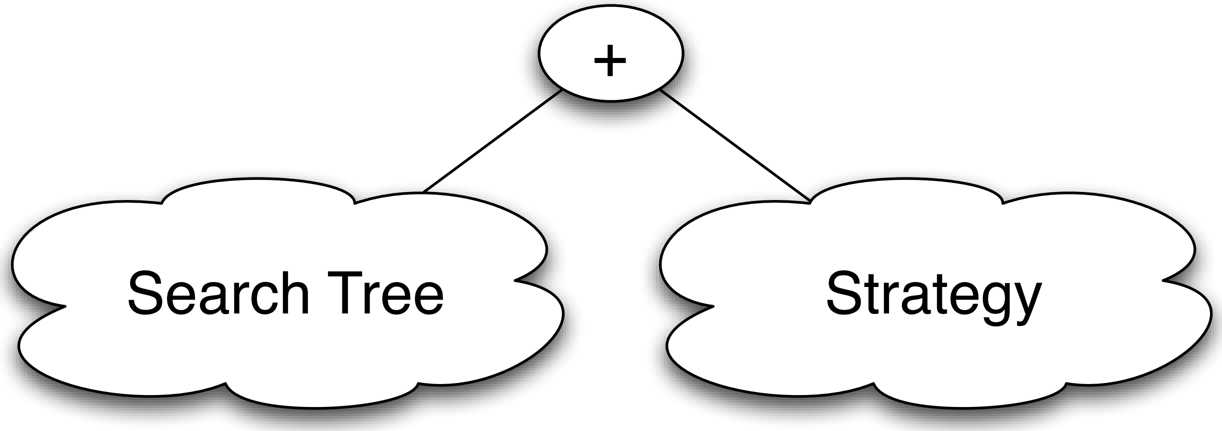
```
import cotfd;
Solver<CP> m();
int n = 20;
range D = 0..n-1;
var<CP>{int} s[D](m,D);
solve<m> {
  forall(k in D)
    m.post(s[k] == sum(i in D) (s[i]==k));
  m.post(sum(k in D) (k-1)*s[k]==0);
}

cout << s << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail    = " << m.getNFail() << endl;
```

Searching!

- Purpose
  - Write your own search procedure
  - Exploit problem semantics for...
    - Variables ordering
    - Value ordering
    - Dynamic symmetry breaking
    - Multi-phase searches
    - Dichotomic branching
    - ....

Search anatomy

- Two pieces
  - Specify a search tree
    - What does the tree look like?
      - variable ordering
      - value ordering
  - Specify [optional] a search strategy



Example with Queens

- Rationale
  - Simple problem
  - Illustrates the techniques
  - Start off with default strategy (DFS)

The basic model

```
import cotfd;

int t0 = System.getCPUTime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);

solve<m> {
   m.post(alldifferent(all(i in S) q[i] + i));
   m.post(alldifferent(all(i in S) q[i] - i));
   m.post(alldifferent(q));
}






cout << "Time     = " << System.getCPUTime() - t0 << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail    = " << m.getNFail() << endl;
```

Finding all solutions

```
import cotfd;

int t0 = System.getCPUTime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);

solveall<m> {
   m.post(alldifferent(all(i in S) q[i] + i));
   m.post(alldifferent(all(i in S) q[i] - i));
   m.post(alldifferent(q));
}






cout << "Time     = " << System.getCPUTime() - t0 << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail    = " << m.getNFail() << endl;
```

Printing and Counting solutions...

```
import cotfd;

int t0 = System.getCPUTime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);
Integer c(0);
solveall<m> {
   m.post(alldifferent(all(i in S) q[i] + i));
   m.post(alldifferent(all(i in S) q[i] - i));
   m.post(alldifferent(q));
} using {
   labelFF(m);
   cout << q << endl;
   c := c + 1;
}
cout << "Nb       = " << c << endl;
cout << "Time     = " << System.getCPUTime() - t0 << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail    = " << m.getNFail() << endl;
```

What is labelFF ?

- The default search procedure...
  - Implements first-fail principle
    - First the variable with the smallest domain
    - Try values in increasing order
- Can’t we write this *ourselves?*

Sure!
Let’s start with a very naive search...
...and build up!

Static Ordering [a.k.a. the label function]

- Simple idea
  - Label variables in their “natural” order (order of declaration)
  - Try values in increasing order

```
...
} using {
  forall(i in S)
    tryall<m>(v in S)
      m.post(q[i] == v);
}
```

Static Ordering 2

- First improvement
  - Skip over variables that are already bound!

```
...
} using {
  forall(i in S : !q[i].bound())
    tryall<m>(v in S)
      m.post(q[i] == v);
}
```

Static Ordering 3

- Second improvement
  - Skip values that are no longer in the domain!

```
...
} using {
  forall(i in S : !q[i].bound())
    tryall<m>(v in S : q[i].memberOf(v))
      m.post(q[i] == v);
}
```

Dynamic Ordering

- First consider the variables with the smallest domain
  - Note that this is dynamic, the domain size changes each time!

```
...
} using {
  forall(i in S : !q[i].bound()) by (q[i].getSize())
    tryall<m>(v in S : q[i].memberOf(v))
      m.post(q[i] == v);
}
```

Dynamic Ordering

- Finally...
  - When we fail, remember that the value is no longer legal!

```
...
} using {
  forall(i in S : !q[i].bound()) by (q[i].getSize())
    tryall<m>(v in S : q[i].memberOf(v))
      m.post(q[i] == v);
    onFailure m.post(q[i]!=v);
}
```

Tweaks...

- Use lighter branching method
  - replace m.post(x[i] == v) by m.label(x[i],v);
  - replace m.post(x[i] != v) by m.diff(x[i],v);
- Light api...

Tweaks...

- Use lighter branching method
  - replace m.post(x[i] == v) by m.label(x[i],v);
  - replace m.post(x[i] != v) by m.diff(x[i],v);
- Light api...

```
class Solver<CP> {
  ...
  Outcome<CP> label(var<CP>{int} x,int v);
  Outcome<CP> diff(var<CP>{int} x,int v);
  Outcome<CP> lthen(var<CP>{int} x,int v);
  Outcome<CP> gthen(var<CP>{int} x,int v);
  Outcome<CP> inside(var<CP>{int} x,set{int} s);
  Outcome<CP> outside(var<CP>{int} x,set{int} s);
  ...
```

Final version

- First-fail principle is 4 lines of code.
- Advantage?
  - You can instrument / modify to your heart's content

```
...
} using {
  forall(i in S : !q[i].bound()) by (q[i].getSize())
    tryall<m>(v in S : q[i].memberOf(v))
      m.label(q[i],v);
    onFailure m.diff(q[i],v);
}
```