An Introduction to Constraint (Logic) Programming Using ECL^{i}PS^e

Stefano Benedettini

DEIS - Dipartimento di Elettronica Informatica e Sistemistica
University Bologna, Second Faculty of Engineering

Artificial Intelligence 2010

Outline

Topics

- Constraint Logic Programming (CLP)
- ECL^{i}PS^e constraint solver
- Search methods for CLP (hints)

Objectives

- Re-discover Prolog as a constraint solver
- Familiarize with ECL^{i}PS^e and its extensions to standard Prolog
- Get a grasp of Constraint Logic Programming
  - Learn how to model problems in ECL^{i}PS^e
  - Use constraint libraries
- . . . have fun with the solver

Objectives

- Re-discover Prolog as a constraint solver
- Familiarize with ECL^{i}PS^e and its extensions to standard Prolog
- Get a grasp of Constraint Logic Programming
  - Learn how to model problems in ECL^{i}PS^e
  - Use constraint libraries
- ...have fun with the solver

Method

- Laboratory lessons will follow a hands-on approach
- During lessons you are supposed to experiment with tools and techniques
  - Few novel theoretical concepts will be introduced
- In order to make the most out of these lessons you should:
  - Have a fair theoretical background (i.e., study stuff you saw during classes)
  - Explore new technologies on your own (documentations, tutorials, websites)
  - Be productive as soon as possible

Method

- Laboratory lessons will follow a hands-on approach
- During lessons you are supposed to experiment with tools and techniques
  - Few novel theoretical concepts will be introduced
- In order to make the most out of these lessons you should:
  - Have a fair theoretical background (i.e., study stuff you saw during classes)
  - Explore new technologies on your own (documentations, tutorials, websites)
  - Be productive as soon as possible
  - . . . anything else?

Method

- Laboratory lessons will follow a hands-on approach
- During lessons you are supposed to experiment with tools and techniques
  - Few novel theoretical concepts will be introduced
- In order to make the most out of these lessons you should:
  - Have a fair theoretical background (i.e., study stuff you saw during classes)
  - Explore new technologies on your own (documentations, tutorials, websites)
  - Be productive as soon as possible
- Ah, of course! You should have fun!

Outline

1. Prolog as a Constraint Solver
2. ECLiPSe Extensions to Prolog
   - Syntactic Facilities
   - Iteration Constructs
   - Modules
3. Constraint Satisfaction with ECLiPSe
   - The Interval Constraints Library: `ic`
   - Search and Optimization with ECLiPSe
   - Other Constraint Libraries

Premises

- During this course a good understanding of Prolog will be assumed
  - You all followed Viroli's course so you'll probably know Prolog better than me anyway. . .
- We will use fancy stuff like custom operators and structures
- We won't use fancier stuff like metaprogramming
  - Though you're free to explore by yourselves

Tools of the Trade

- At this point any Prolog implementation will do
- But we will abandon the usual Prolog fairly soon
- You'd better start with ECL^{i}PS^e right away

`http://eclipse-clp.org/`

Resources for ECL^{i}PS^e

1. ECL^{i}PS^e main site `http://eclipse-clp.org/`
2. `http://eclipse-clp.org/examples`
3. Keep your friends close but your documentation closer:
   - `http://eclipse-clp.org/doc` (we will use that in due time)
4. Introductory book on CP using ECL^{i}PS^e: Constraint Logic Programming Using ECL^{i}PS^e

Resources for Constraint Programming in General

- CS problems and models in various languages: `http://www.csplib.org/`
- Gecode: THE C++ library for CP (not for the faint of heart)
- Guido Tack's Ph.D. dissertation (for a challenge)
- Java library: `http://jacop.osolpro.com/`
- Python library (proprietary): `http://www.eveutilities.com/products/emma`
- Python library (GPL): `http://labix.org/python-constraint`

Resources for Constraint Programming in General

- Comet language: `http://dynadec.com/support/downloads/` (keep an eye on this)
- On-line guide to CP by Roman Bartak
  - ...and much more!
- A blog on CP (serious stuff)

Mapping CSPs in Prolog

What we would like to do

- Setup variables and domains
- State constraints
- Specify a search strategy

What Prolog has to offer

- Logic variables and closed world assumption
- Predicates can be seen as very basic form of constraint
- Backtracking for free

What Prolog lacks

- Constraint propagation $\Rightarrow$ Efficiency!
- Constraints and variables are not first-class citizens

Constraints in Prolog

- Any predicate is really a constraint

Example

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

p(X, X).
p(X, 3).
```

```
:- member(X, [1, 2, 3, 4]), X > 2.
```

yields X = 3, X = 4, while

```
:- member(X, [1, 2, 3, 4]), member(Y, [2, 3, 4, 5]),
    p(X, Y).
```

yields (X, Y) = (1, 3), (2, 2), (2, 3), (3, 3), (3, 3), (4, 3), (4, 4)

Do Constraints in Prolog Really Work. . . ?

Example

How about these?

```
:- X > 3, X < 2. % clearly inconsistent
:- 4 is X + 3. % clearly X = 1
```

Both raise an "Instantiation fault" error

- Non-logical predicates and closed world assumption are Prolog greatest weaknesses
- Variables in arithmetic predicates must be instantiated (else error)
- For which values the second query is satisfied? For none, because you never told Prolog that 1 + 3 is true

Do Constraints in Prolog Really Work. . . ?

Example

How about these?

```
:- X > 3, X < 2. % clearly inconsistent
:- 4 is X + 3. % clearly X = 1
```

Both raise an “Instantiation fault” error

- Non-logical predicates and closed world assumption are Prolog greatest weaknesses
- Variables in arithmetic predicates must be instantiated (else error)
- For which values the second query is satisfied? For none, because you never told Prolog that 1 + 3 is true

. . . The Answer Is: "So So"

Consequences

- Constraints can only be tested for satisfiability
  - There is no "constraint propagation" (domain reduction)
- Prolog natively supports only generate and test!

CSP in Prolog

```
solve_problem(Variables) :-
  declare_domains(Variables),
  search(Variables),
  test_constraints(Variables).
```

- First I generate a complete assignment to the decision variables
- Then I test satisfiability

The Simplest Prolog Constraint Solver That Can Possibly Work

```
% Define domains for a list of variables
domain([], _, []).
domain([H|T], D, [H dom D|Rest]) :- domain(T, D, Rest
    ).

% Define integral domains for a list of variables
ints(Xs, Min, Max, Vs) :-
  int_range(Min, Max, R),
  domain(Xs, R, Vs).

% Simple labeling
label([]).
label([V dom D|T]) :- member(V, D), label(T).
```

Some Constraints

```
% In constraint
X in D :- member(X, D).

% Not in constraint
_ not_in [].
X not_in [H|T] :- not(X in [H|T]).

% All different
all_different([]).
all_different([H|T]) :-
  all_different(T),
  H not_in T.

% Ordered
ordered([_]).
ordered([X, Y|Z]) :- X < Y, ordered([Y|Z]).
```

Outline

The ECL^{i}PS^e Programming Environment

ECL^{i}PS^e supports a number of extensions that facilitate Prolog programming:

1. Structures and arrays
2. "Function-like" predicates
3. Iteration constructs
4. Modules

For more information, please refer to the User Manual

ECL^{i}PS^e is a mature piece of software. We won't have time to cover everything. You are encouraged to explore by yourselves.

Outline

"Function-like" Predicates

- Unofficial name to refer to a syntactic facility that allows you to use custom predicates in expression context
  - "Expression Context" is wherever ECL^{i}PS^e expects arithmetic operations that are subsequently evaluated
  - Usually indicated with metavariable *Expr* in documentation
- You can use a n-ary predicate $p(X_1, \ldots, X_n)$ as a function that expects $X_1, \ldots, X_{n-1}$ as arguments and returns X_n as result

Example

```
p(_, 3).

summation([], 0).
summation([H|T], S) :- summation(T, S1), S is S1 + H.

:- 10 is 7 + p(ciao).
:- 19 < summation([1, 2, 3]) + 15.
```

Structure Notation

- An easy way to declare and access arguments of a Prolog structure
  - Declare a structure using:
    `:- local struct(`*`functorName`*`(`*`fieldNames...`*`)).`
  - Can refer to fields by name instead of index
  - Use *`fieldName`* `of` *`structName`* to obtain field index

Example

Declaring and using a structure

```
:- local struct(student(name, surname, exam, mark)).

:- student{name:john, surname:doe} = student(N, SN, _
    , _), N = john, SN = doe.
:- S = student{name:jane}, arg(4, S, 30), S = student
    (jane, _, _, M), M = 30.
```

Enumerations

- ECL^{i}PS^e does not support directly enumerations, but they can be emulated via structures
  - Encode each value of your enumeration with an integer
  - Declare a structure whose field names are the enumeration values
  - Use operator `of` to map a value to an integer
  - You should never instantiate a structure of that kind
  - The inverse mapping is not as easy

Example

A color example

```
:- local struct(colors(black, white, red, green, blue
    )).

:- X = black of colors, X = 1
:- X is (white of colors) + (red of colors), X = 5 %
    useless but still
```

Arrays

- Arrays in ECL^{i}PS^e are really a syntactic convenience to indicate structures of structures
- Functor name is irrelevant
  - `matrix(row(1, 2, 3), row(4, 5, 6))` is a 2×3 array and so is `a(b(1, 2, 3), c(4, 5, 6))`
- `dim(A, D)` unifies a new array `A` with functor name `[]` and its dimensions `D`
- Use `subscript/3` to access individual elements or array slices
- Usual C-like subscription operator available in expression context provides same behaviour as `subscript/3`

Example

```
:- dim(A, [2, 3]), A = [](([](_, _, _), [](_, _, _))
:- dim([]([](1, 2), [](3, 4)), D), D = [2, 2]
:- dim(A, [2, 3]), subscript(A, [2, 1..3], R1), R2 is
     A[2, 1..3], R1 = R2
```

Outline

Why Iteration in Prolog?

- Although every form of iteration can be expressed by a recursion, this bring some disadvantages:

  Code bloat: every iteration requires you to write an additional predicate

  Readability: those predicates are coded far away in your code and impact readability

  Non-locality: if your iteration has to access many local variables, you have to write a predicate with that many parameters

- Iteration in ECL^{i}PS^e is as powerful as recursion and often more concise

Iteration Constructs Generalities

- All iteration constructs have the form `(`*`IterationSpecifiers`*` do `*`Body`*`)`
- *IterationSpecifiers* is a comma-separated sequence of iteration specifiers
- Each specifier introduces an iteration variable visible only in *Body*
- *Body* can access only variables provided by iteration specifiers (but see `param`)
- Each iteration step, iterations variables in *Body* are unified with iteration values
- All iteration specifiers step in parallel and the whole loop stops when one of the specifiers completes

Iteration Constructs: `foreach`

- `(foreach(E, List) do Body)` executes goal `Body` iteratively unifying `E` to each element in list `List`
- Has invertible semantics: can be used to scan a list or to construct a list

Example

```
:- (foreach(E, [1, 2, 3]) do writeln(E)). % prints 1
   2 3
:- (foreach(X, L), foreach(E, [10, 20, 30]) do X is E
    // 10), L = [1, 2, 3].
```

Iteration Constructs: `param` and `count`

- `param(`*`Variables`*`...)` makes *Variables* visible inside iteration body
- `count(`*`Index`*`, `*`MinExpr`*`, `*`MaxExpr`*`)` unifies *Index* to each integer in [*MinExpr*, *MaxExpr*]

Example

```
multiply(L, K, Out) :-
  (foreach(E, L), foreach(X, Out), param(K) do X is K
      * E).

my_length(L, X) :- % same as length/2 builtin
  (foreach(_, L), count(_, 1, X) do true).

:- (count(I, 1, 4) do writeln(I)). % prints 1 2 3 4
```

Iteration Constructs: `foreacharg` and `foreachelem`

- `foreacharg/2` iterates on a structure/array arguments
- `foreachelem/2` works only for arrays (functor name must be `[]`). Flattens the array and iterate on its elements
- Both support an optional third argument that is unified to element index

Example

```
:- M = []([](1, 2, 3), [](4, 5, 6)), (foreachelem(E,
    M) do writeln(E)), % prints 1 2 3 4 5 6
(foreacharg(Row, M) do writeln(Row)), % prints [](1,
    2, 3) [](4, 5, 6)
(foreachelem(E, M, [I, J]) do writeln(I-J-E)), %
    prints 1-1-1 1-2-2 1-3-3 2-1-4 2-2-5 2-3-6
(foreacharg(R, M, I) do writeln(I-R)). % prints
    1-[](1, 2, 3) 2-[](4, 5, 6)
```

Iteration Constructs: `for` and `multifor`

- `for(`*`Index`*`, `*`MinExpr`*`, `*`MaxExpr`*`)` work like its imperative counterpart
- `multifor(`*`IndexList`*`, `*`MinList`*`, `*`MaxList`*`)` concisely describes multiply nested `for`s
- Both constructs support an optional fourth argument to provide an iteration step

Example

```
evens(L) :-
  (for(I, 2, 100, 2), foreach(I, L) do true).

mutliplication_table(T) :-
  dim(T, [10, 10]),
  (multifor([X, Y], [1, 1], [10, 10]), % also multifor
       ([X, Y], 1, 10)
    foreachelem(E, T) do
      E is X * Y
  ).
```

Iteration Constructs: `fromto`

- Most general construct
- `(fromto(`*`Init`*`, `*`In`*`, `*`Out`*`, `*`Last`*`) do `*`Body`*`)` introduces local variables *In* and *Out* in *Body* and behaves as follows:
  1. First *`Init = In`*
  2. Executes iteration body
  3. After each iteration checks whether *`Last = Out`* and breaks if unification succeeds
  4. Otherwise unifies a fresh *In* variable with *Out* and repeats from step 2

Example

```
filter_even(L, F) :-
  (foreach(E, L), fromto(L, In, Out, []) do
    (0 is E rem 2 -> In = [E|T], Out = T; In = Out)
  ).

reverse(L, R) :-
  (foreach(E, L), fromto([], In, [E|In], R) do true).
```

Outline

ECL^{i}PS^e Modules in a Nutshell

Advantages

- Group functionally related predicates in the same place
  - Usually one module for each file
- Control naming access
- Modules form a namespace structure

Using modules

- use `lib(`*`ModuleName`*`)` to compile an ECL^{i}PS^e library and import exported names
- use `use_module(ModuleName)` to compile a module and import its exported names
- use `:` operator to disambiguate a name: *`ModuleName`*`:(`*`predicate`*`)`

ECL^{i}PS^e Modules in a Nutshell

Writing modules

- use directive `module(ModuleName)` at the top of a source file to define a module
- `export` directive states which predicates to export

Example

```
:- module(my_little_module).
:- export print_matrix/1.

print_row(Row) :-
  (foreachelem(E, Row) do write(E), write(" ")),
  nl.

print_matrix(M) :-
  (foreacharg(Row, M), param(M) do print_row(Row)).
```

`print_matrix` is available when this module is imported but `print_row` is not.

Outline

Overcoming Prolog Limitations

The Disease

- Variables in arithmetic constraints must be instantiated
- Constraint predicates are mostly used for testing and cannot instantiate variables except the simplest like `or`, `and`, etc...
- No constraint propagation

The Cure

- Every goal (i.e., constraint) that is not fully instantiated is suspended and put into the constraint store
- Propagation is applied to variable domains at different strength

Before the Cure

Example

```
ordered_list([_]).
ordered_list([X, Y|T]) :- X < Y, ordered_list([Y|T]).

:- ordered_list([1, 2, 3, 4, 5]). % works as expected
:- ordered_list([1, X, 3, Y, 5]). % we suppose X, Y
    be integral variables
```

The second goal raises an instantiation error.

After the Cure

Example

```
:- lib(ic). % load a solver library

ordered_list([_]).
ordered_list([X, Y|T]) :- X #< Y, ordered_list([Y|T])
    .

:- ordered_list([1, X, 3, Y, 5]).
:- ordered_list([1, X, Y, 5]).
```

In the first case ECL^{i}PS^e enforces variable integrality and correctly propagates the constraints $1 < X < 3$ and $3 < Y < 5$ thereby inferring $X = 2, Y = 4$.

In the second case we have a delayed goal, that is a goal that cannot be solved at the moment. ECL^{i}PS^e propagates inequalities and infers $X \in 2, 3, Y \in 3, 4$.

How to Write a Problem Model

CSP in ECL^{i}PS^e

```
solve_problem(Variables) :-
  declare_domains(Variables),
  setup_constraints(Variables),
  search(Variables).
```

How to Write a Problem Model

CSP in ECL^{i}PS^e

```
solve_problem(Variables) :-
  declare_domains(Variables),
  setup_constraints(Variables),
  search(Variables).
```

This part comprises model definition.

How to Write a Problem Model

CSP in ECL^{i}PS^e

```
solve_problem(Variables) :-
  declare_domains(Variables),
  setup_constraints(Variables),
  search(Variables).
```

This is the actual search.

How to Write a Problem Model

CSP in ECL^{i}PS^e

```
solve_problem(Variables) :-
  declare_domains(Variables),
  setup_constraints(Variables),
  search(Variables).
```

Modeling and search are independent phases in the resolution of a problem.

How to Write a Problem Model

CSP in ECL^{i}PS^e

```
solve_problem(Variables) :-
  declare_domains(Variables),
  setup_constraints(Variables),
  search(Variables).
```

Problem model and choice points

- *Do not* leave choice points during constraint setup!
- If you do, the solver will miss an opportunity do constraint propagation
- You are actually splitting a CSP into multiple CSPs, making choices *before* propagation, not *after*
- But it *doesn't* mean you can't use Prolog non determinism

An Overview of the Libraries

- We'll mostly use only the more recent `ic` library and briefly touch its derivatives:

  `ic_global` very useful global constraints

  `ic_sets` supports set variables

  `ic_symbolic` handles (not so) conveniently symbolic domains

- Don't use old `fd` libraries: stick with `ic`!
- Basic searching strategies and B&B optimization
- *Many* more constraints available (look them up in the documentation):
  - Bin packing
  - Knapsack
  - Scheduling
- Generalized Propagation and Constraint Handling Rules (CHR) will be barely mentioned

Outline

Introduction

- Powerful hybrid solver that works on (intervals of) integers and reals
- Variable domains are unions of disjoint intervals
- It enforces bound consistency:
  - Weaker than arc-consistency
  - Iteratively "squeezes" a variable domain: $x, y \in [0, 5] \cap \mathbb{Z}, x < y \Rightarrow x \in [0, 4], y \in [1, 5]$
  - Cannot propagate "holes"
- Handles linear, arithmetic and non-linear constraints
- Issues with real variables:
  - How do we cope with limited precision?
  - How do we represent "holes" in a real domain?
  - Does this make sense: $x \in [0, 1], x \neq 0.5$?

Variable Domains

- Constrains integer as well real domains with `::`, `$::` and `#::` operators:
  - *`ListOfVariables :: RangeSpecification`* specify either an integer or a real domain depending on the range specification
  - `$::` enforces real domains
  - `#::` enforces integer domain
  - I suggest to employ only `#::` or `$::` to clearly state variable type
- These operators work also with multidimensional arrays
- See documentation for further details

Example

```
:- Var #:: [1..5, 7, 9..12].
:- [X, Y] #:: 1..5.
:- dim(A, [3, 2]), A :: 0..1.0.
```

Arithmetic Constraints

- `ic` support every arithmetic and comparison operator of standard Prolog
  - qualified use: `ic:(X` *op* `Y)` where *op* is one of >, <, `=<`, `=:=`, `=\=`, `+`, `-`, `*`, ...
  - Prepend $ to comparison predicates to avoid qualification: $>, $<, $=<, $=, $\=, ...
  - Prepend # to comparison predicates to also enforce integrality: #>, #<, #=<, #=, #\=, ...
- New variables spawn into existence as soon they are used

Example

```
:- [X, Y] :: 1..5.0, Z #= X + 1, ic:(Z < Y). % 1
:- length(V, 4), V :: 0..5, (fromto(V, [X, Y|T], [Y|T
    ], [_]) do ic:(X < Y)). % 2
```

1. $X \in [1,3] \cap \mathbb{N}, Y \in [2,5], Z \in [2,4] \cap \mathbb{N}$
2. $\forall i \in [1,4]\ x_i \in \mathbb{N}, x_1 \in [0,2], x_2 \in [1,3], x_3 \in [2,4], x_4 \in [3,5]$

Boolean Constraints

- `ic` supports basic boolean operators:
  - `or`, `and`, `xor`, `=>` all infix and `neg` (for negation)
- They automatically enforce a boolean domain, that is $\{0, 1\}$

Example

How to solve a SAT without really trying

```
:- X and (Y or Z) and (X or neg Y)
:- X and neg X
```

1. Infers $X = 1$
2. Infers inconsistency

Reified Constraints

- How can we conditionally enforce a constraint?
  - How can I express if $x \in [1, 3]$ then $y < z$?
  - Remember that we cannot leave choice points so, for instance, operator `->` is out of question
- Reified constraints come to the rescue
- Domain, comparison and boolean constraints all have a reified version
  - If the original constraint is op`(X, Y)`, its reified version is op`(X, Y, B)` where B is a boolean variable
  - If $B = 1$ the constraint holds, otherwise holds its negation
- Thanks to "function-like predicate" facility we can "chain" reified constraints in expression context

```
:- #<(X, Y, B1), #::(X, 1..5, B2), B1 or B2.
```

becomes

```
:- (X #< Y) or (X #:: 1..5).
```

Global Constraints and Propagation Strength

- Most of constraints in `ic` are binary
  - Also `alldifferent` is nothing more than manually setting pairwise inequalities
- We need a better way to model global constraints than separating them in binary constraints
  - Propagation strength is lower
  - That is we cannot rule out as many values as a we could
- Enter `ic_global`
  - Lots of high level constraints:
    - Minimum/maximum of a list
    - `occurrences`
    - a stronger `alldifferent`
    - `ordered` (useful for symmetry breaking)
- Also check `ac_eq` in `ic`
  - Provides an arc-consistent (and not only bound consistent) version of $X \ \#= \ Y \ + \ C$ (C is an integer constant)

An Example with Global Constraints

```
:- length(V, 100), V #:: 1..99, ic:alldifferent(V).
:- length(V, 100), V #:: 1..99, ic_global:
    alldifferent(V).
:- length(V, 5), V #:: 1..5, ordered(<, V).
:- [X, Y] #:: 1..10, X #\= 7, X #= Y + 2.
:- [X, Y] #:: 1..10, X #\= 7, ac_eq(X, Y, 2).
```

1. Clearly inconsistent but no propagation is performed
2. Immediately returns “No” as answer
3. Correctly infers, $X_i = i, i = 1, \ldots, 5$
4. $X \in [3, 6] \cup [8, 10]$ and $Y \in [1, 8]$ but cannot propagate the “hole” at $X = 7$
5. $X \in [3, 6] \cup [8, 10]$ and $Y \in [1, 4] \cup [6, 8]$: now it is arc consistent

Outline

Labeling

- Propagation is insufficient to prove satisfiability in all but the simplest CSPs
- In ECL^{i}PS^e this translate into having delayed goals
- The simples strategy consists in:
  1. Recursively considering all unassigned variables in some order
  2. Try all values in its domain in ascending order
  3. Return a feasible assignment or backtrack and try another value
- this is called labeling

Example

Pythagorean Triplets

```
:- V = [A, B, C], V #:: 1..100, ordered(<, V), A^2 + B^2
     #= C^2, findall(V, labeling(V), L).
```

- There are 52 unique Pythagorean triplets of integers in $[1, 100]$
- Note the use of `ordered` to remove symmetries

Fine Tuning Your Search

- ECL^{i}PS^e provides a general mechanism to customize a search in the form of predicate `ic:search`
- You must specify:
  1. Variable selection strategy
  2. Value selection strategy
- Some heuristics are already defined:

  Variable selection: input order, (anti) first-fail, most-constrai ...

  Value selection: ascending order, starting from max/middle/min value, ...
- Search strategy can have a huge impact on efficiency

Optimization by Branch-and-Bound

- ECL^{i}PS^e comes also with a ready-made B&B implementation to tackle constraint optimization problems
- You must also provide an objective function f
- ECL^{i}PS^e B&B works as follows (for minimization problems):
  1. Generates a feasible solution s and impose the constraint $f(s') < f(s)$ for the remainder of the search
  2. Restarts or continues the search

Example

Which is the largest integer $n < 1000$ such that the cube of the sum of its digits equals the number itself?

```
:- [X, Y, Z] #:: 0..9,
  N #= 100 * X + 10 * Y + Z,
  C #= (-N),
  (X + Y + Z)^3 #= N,
  minimize(labeling([X, Y, Z]), C).
```

Optimization by Branch-and-Bound

- ECL^{i}PS^e comes also with a ready-made B&B implementation to tackle constraint optimization problems
- You must also provide an objective function f
- ECL^{i}PS^e B&B works as follows (for minimization problems):
  1. Generates a feasible solution s and impose the constraint $f(s') < f(s)$ for the remainder of the search
  2. Restarts or continues the search

Example

Convert a maximization into a minimization problem.

```
:- [X, Y, Z] #:: 0..9,
   N #= 100 * X + 10 * Y + Z,
   C #= (-N),
   (X + Y + Z)^3 #= N,
   minimize(labeling([X, Y, Z]), C).
```

Optimization by Branch-and-Bound

- ECL^{i}PS^e comes also with a ready-made B&B implementation to tackle constraint optimization problems
- You must also provide an objective function f
- ECL^{i}PS^e B&B works as follows (for minimization problems):
  1. Generates a feasible solution s and impose the constraint $f(s') < f(s)$ for the remainder of the search
  2. Restarts or continues the search

Example

Which is the largest integer $n < 1000$ such that the cube of the sum of its digits equals the number itself? It's 512.

```
:- [X, Y, Z] #:: 0..9,
  N #= 100 * X + 10 * Y + Z,
  C #= (-N),
  (X + Y + Z)^3 #= N,
  minimize(labeling([X, Y, Z]), C).
```

Outline

Symbolic Domains With `ic_symbolic`

- Sometimes it's more natural to work with a finite domain other than (an interval of) integers
  - Colors
  - Days of the week
- `ic_symbolic` permits this kind of symbolic domains
- This library is implemented on top of ic
- Every value in a symbolic domain is mapped to an integer
- Define a new symbolic domain with `domain` declaration
  `:- local domain(`*`domainName`*`(`*`values`*`...)).`

Example

```
:- local domain(weekday(mo, tu, we, th, fr, sa, su)).

:- X &:: weekday, X &\= tu, X &< fr.
```

Note the implicit ordering of values.

Set Variables With `ic_sets`

- This library manipulates integer set variables
- A set is simply a sorted list of integers without repetitions
- Much like `ic`, it provides bound consistency on set variable domains
- But what is a "range of sets"?
- A set variable σ is a triple: $\langle S_{lb}, S_{ub}, C \rangle$
  - S_{lb} is a set containing all the elements which are certainly in σ
  - S_{ub} is a set containing all the elements that might be in σ (obviously $S_{lb} \subseteq S_{ub}$)
  - C is an integer variable equal to $|\sigma|$
- See also the concept of Lattice

Example

```
:- X :: [1]..[1, 2, 3], insetdomain(X, _, _, _).
```

Finds: $\langle \{1, 2, 3\}, \{1, 2\}, \{1, 3\}, \{1\} \rangle$