Constraint Satisfaction Problems

The ECL^iPS^e Constraint Logic Programming System

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Christian Becker-Asano, **Bernhard Nebel** and **Stefan Wölfl**

January 26, 2015

Motivation

What is *ECLiPSe* ?

"*ECL*^{i}*PS*^{e} is designed for solving combinatorial optimization problems, for the development of new constraint solver technology and their hybrids, and for the teaching of modelling, solving and search techniques."

(`http://sourceforge.net/projects/eclipse-clp/`)

What is ECL^iPS^e ?

ECL^iPS^e :

1. is a declarative language
2. is a Constraint Programming toolkit
3. is available open source (Mozilla Public License)
4. has been applied to areas of planning, scheduling, resource allocation, timetabling, transport...
5. provides bindings for Python, C/C++, Tcl/Tk, Java (Eclipse IDE plugin), SQL, ...
6. can be interfaced to remotely (RPC)

Why ECL^iPS^e ?

The TPK (Trabb Pardo-Knuth) algorithm of Donald Knuth and Luis Trabb Pardo (1976) (the “Hello World!” of algorithms):

“The algorithm prompts for 11 real numbers ($a_0 \ldots a_{10}$) and for each a_i computes $b_i = f(a_i)$, where $f(t) = \sqrt{|t|} + 5t^3$. After that for $i = 10 \ldots 0$ (in that order) the algorithm outputs a pair (i, b_i) if $b_i \leq 400$, or $(i, \mathtt{TOO\ LARGE})$ otherwise.” (Dymchenko & Mykhailova, 2014)

Declarative programming with logic

The TPK algorithm in ECL^iPS^e / Prolog:

```
f(T, Y) :-
    Y is sqrt(abs(T)) + 5*T^3.
main :-
    read(As),
    length(As, N), reverse(As, Rs),
    ( foreach(Ai, Rs), for(I, N - 1, 0, -1) do
        Bi is f(Ai),
        ( Bi > 400 ->
            printf("%w TOO LARGE\n", I)
        ;
            printf("%w %w\n", [I, Bi])
        )
    ).
```

(Dymchenko & Mykhailova, 2014)

Declarative programming with functions

The TPK algorithm in python 3:

```
>>> def f(x): return abs(x) ** 0.5 + 5 * x**3

>>> print(', '.join('%s:%s' %
        (x, v if v<=400 else "TOO LARGE!")
        for x,v in ((y, f(float(y)))
                    for y in input('\nnumbers: ')
                    .strip.split()[:11][::-1])))
```

(http://rosettacode.org/wiki/Trabb_Pardo%E2%80%93Knuth_algorithm#Python)

Is the logical Prolog version “easier/better” than the functional Python version?

Python binding for *ECL^{i}PS^e*

The “Search Solution” example of `pyclp`:

```
from pyclp import *
init() # Init ECLiPSe engine
Compound("lib",Atom("ic")).post_goal() # Load ic library
A_var=Var() # Create variable A
B_var=Var() # Create variable B
# [A,B]#::1..10
Compound("#::",PList([A_var,B_var]),\
         Compound("..",1,10)).post_goal()
Compound("#<",A_var,B_var).post_goal() # A#<B
Compound("#=",A_var,5).post_goal() # A#=5
# labeling([A,B])
Compound("labeling",\
         PList([A_var,B_var])).post_goal()
# Loop on all solution and print them.
while (resume()[0]==SUCCEED):
    print(B_var)
    # backtracking over solutions
    Atom("fail").post_goal()
cleanup() # Shutdown ECLiPSe engine
```

So again, why *ECL^{i}PS^e* ?

$\Rightarrow$ Because it can be considered more elegant and even easier to (only) `declare` a problem logically and then use all the power of standard `reasoning` and `search` algorithms to find a solution.

ECL^{i}PS^e contains all major CLP algorithms as libraries and provides the full power of the Prolog language for problem modeling.

Suitable problems for ECL^iPS^e

Characteristics of problems suitable for ECL^iPS^e :

1. There are no general methods or algorithms
   - NP-completeness
   - Different strategies and heuristics have to be tested
2. Requirements are quickly changing
   - Programs should be flexible enough to adapt
3. Decision support required
   - Co-operate with user
   - Friendly interfaces

Problem modeling

Logic programming

Constraint Programming can be characterized by two pseudo-equations:

$$\mathit{Solution} = \mathit{Logic} + \mathit{Control} \tag{1}$$

$$\mathit{Control} = \mathit{Reasoning} + \mathit{Search} \tag{2}$$

Equation (1): a `solution` can be found by:

1. a `logical`, declarative description of the problem, and
2. `control` information for the computer to deduce it.

Equation (2): `control` is a combination of:

1. `reasoning` to (efficiently) limit the search space, and
2. subsequent (inefficient) `search` through that space

Problem modeling deals with the `Logic` part of Equation (1).

Issues in Problem Modeling

A good formalism should fulfill the following criteria:

1. Expressive power:
   formal model of real world problem possible?
2. Clarity for humans:
   ease of use of formalism (read, write, understand, modify)
3. Solvability for computers:
   Good methods available to solve problem?

Higher-level models

- \+ closer to the user and the problem
- \+ easier to understand and trust, to debug and modify, but
- \- difficult to see how they can be solved

Issues in Problem Modeling

Classical source of error in application development:
$\Rightarrow$ Transition from `formal description` to `final program`
$\Rightarrow$ Can the final program be trusted?

CLP solution:

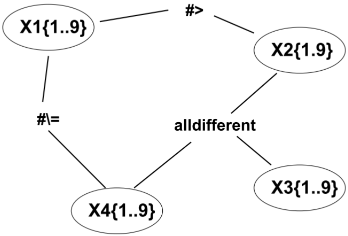
- Keep initial formal model as part of the final program
- Enhance rather than rewrite:
  - Add control annotations (e.g., algorithmic or heuristic information)
  - Transform `higher-level` (problem) constraints into `low-level` (solver) constraints

Built-in language constructs used in modeling:

- **Build-in constraints**: `X #> Y`
- **Abstraction**:
```
bef(task(Si,Di,),task(Sj,Dj)) :- Si+Dj #<= Sj.
```
- **Conjunction**: `betw(X,Y,Z) :- X #< Y, Y #< Z.`
- **Disjunction**: `neighb(X,Y) :- ( X #= Y+1 ; Y #= X+1 ).`
- **Iteration**:
```
not_among(X,L) :-
( foreach(Y,L), param(X) do X #\= Y ).
```
- **Recursion**:
```
not_among(X,[]).
not_among(X,[Y|Ys]) :- X #\= Y, not_among(X,Ys).
```

CLP and ECL^iPS^e

An example constraint network (Cheadle et al., 2014):



Variables
with attributes
e.g. domain

Constraints
predicates involving
one or more variables

Model
= setup program:

```
[X1,X2,X3,X4]::1..9,
X1 #> X2,
alldifferent([X2,X3,X4]),
X1 #\= X4.
```

But, of course, one problem can be modeled in multiple ways..

Same Problem – Different Model

```
sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
    alldifferent(Digits),
    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.
```

```
sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
    Carries = [C1,C2,C3,C4],
    Carries :: [0..1],
    alldifferent(Digits),
    S #\= 0, M #\= 0,
    C1         #= M,
    C2 + S + M #= O + 10*C1,
    C3 + E + O #= N + 10*C2,
    C4 + N + R #= E + 10*C3,
         D + E #= Y + 10*C4.
```

Modeling rules

Both models work fine, but involve different variables and constraints.

$\Rightarrow$ "Finding good models [..] requires substantial **expertise** and **experience**." (Cheadle et al., 2014)

Declarative model is constraint setup code $\Rightarrow$ should be deterministic and terminating, so **general rules**:

- **Careful with disjunctions**: Don't leave choice points (i.e., alternatives for backtracking); should be deferred until search phase
- **Use only simple conditionals**: Conditions (`...->...;...`) must be true or false at modeling time!
- **Use only structural recursion and loops**: Termination conditions must be known at modeling time!

Preliminary summary

ECL^iPS^e :

- is a declarative constraint programming framework
- interfaces with many programming languages
- is based on the paradigm:
  `Solution = Logic + Control`
- uses Prolog for problem modeling

⇒ Next, introduction to Prolog..

Prolog & ECL^iPS^e

Introduction: Terms and their data types

Prolog data (`terms`) and programs are built from the following data types:

- Numbers
- Strings
- Atoms
- Lists
- Structures

They are introduced next..

Numbers

Numbers in ECL^iPS^e come in several flavors:

1. Integers can be as large as fits into memory, e.g.:
   `123 0 -27 393423874981724`
2. Floating point number (repr. as IEEE double floats), e.g.:
   `0.0 3.141592653589793 6.02e23 -35e-12 -1.0Inf`
3. Also available: rationals and bounded reals

Beware: Performing arithmetic requires **is/2** predicate:

```
?- X is 3 + 4.
X = 7
Yes
```

Predicate **=/2** constructs term corresp. to arithmetic expression:

```
?- X = 3 + 4.
X = 3 + 4
Yes
```

Strings

Strings represent arbitrary sequences of bytes:

```
"I am a string!"
"string with a newline \n and a null \000 character"
```

Strings versus Atoms:

1. many predicates accept both strings and atoms
2. internally, the data types are quite different:
   - `string` stored as character sequence
   - `atom` mapped into internal constant via dictionary table

⟶ Copying and comparing:
   - `atoms` in unit time
   - `strings` in time proportional to string length

⟶ However, recollection of freed dictionary memory needs garbage collection

Strings versus Atoms

Consider the following example:

```
[eclipse 1]: [user].
afather(john, george).
afather(sue, harry).
afather(george, edward).
sfather("john", "george").
sfather("sue", "harry").
sfather("george", "edward").
yes.

[eclipse 2]: afather(sue,X).
X = harry
yes.

[eclipse 3]: sfather("sue",X).
X = "harry"      More? (;)
no (more) solution.
```

⇒ Atoms should always be preferred when they are involved in unification and matching.

Atoms

Atoms are simple symbolic constants:

1. similar to `enumeration` type constants in other languages
2. no special meaning attached to them by the language
3. syntactically:
   - all words starting with a lower case letter are atoms
   - sequences of symbols are atoms
   - anything in single quotes is an atom
4. E.g.: `atom quark i5 -*- ??? 'Atom' 'an atom'`

Lists

Lists are:

1. ordered sequences of (any number of) elements, each of which is itself a term
2. delimited by square brackets (`[ ]`) and its elements comma separated

Examples: `[1,2,3]`, `[berlin, tokyo, freiburg]`, `["csp1415", 42, [1,2,3], freiburg]`

More notation:

1. empty list: `[]`
2. head and tail: `[Head|Tail]`, with
   - `Head` a single element
   - `Tail` a (possibly empty) list

→ Equivalent lists: `[1,2,3]`, `[1|[2,3]]`, `[1|[2|[3]]]`, `[1|[2|[3|[]]]]`

Structures

Structures correspond to structs or records in other languages:

1. always has a name, which looks like an atom
2. aggregates a fixed number of components, i.e. arguments that themselves are terms
3. general structure: *<name>(<arg>$_1$, ...<arg>$_n$)*
4. `arity` is the number of arguments
5. name and arity together is called `functor`, often written as `name/arity`, e.g., `flight/4`

Examples:

```
date(december, 25, "Christmas")
element(hydrogen, composition(1,0))
flight(london, new_york, 12.05, 17.55)
```

Operator syntax (structures)

Prefix, infix, and postfix notation:

1. Unary structures also possible in prefix or postfix notation, e.g., `old berta.` the same as `old(berta).`
2. Binary structures also possible in prefix or infix notation, e.g., `1 plus 5.` the same as `plus(1, 5).`
3. these notations need to be declared with `:- op(+Precedence, +Associativity, ++Name)`
4. if in doubt, use `display/1` to check parsing of term:

```
[eclipse]: display(a+b*c).
+(a, *(b, c))
yes.
```

Summary of data types

1. **Numbers**: ECL^iPS^e has `intergers`, `floats`, `rationals`, and `bounded reals`
2. **Strings**: character sequences in double quotes
3. **Atoms**: symbolic constants, usually lower case or in single quotes
4. **Lists**: constructed from cells that have an arbitrary head and a tail, which is again a (possibly empty) list
5. **Structures**: have a `name` and a certain number (`arity`) of arbitrary arguments, this characteristic is called the `functor`, and written `name/arity`

Predicates

Other programming languages have `procedures` and `functions`

⇒ Prolog and *ECL^{i}PS^e* have `predicates`:

1. a predicate is something that has a truth value
2. a predicate *definition* defines what is `true`
3. a predicate *invocation* or *call* checks its truth value

Goals and queries

Predicate examples for `integer/1`:

```
integer(123)            is true
integer(atom)           is false
integer([1,2])          is false
```

These predicate calls are `goals`.
If supplied by a user as a *starting goal*, the goal becomes a `query`, e.g.:

```
?- integer(123).
Yes.
?- integer(atom).
No.
```

Queries always return either `Yes.` or `No.`

Conjunctions

Goals are often combined to form *conjunctions* (AND) or *disjunctions* (OR).

Conjunctions:

1. are built using commas
2. are only true if all conjuncts are true

Examples:

```
?- integer(5), integer(7), integer(9).
Yes.
?- integer(5), integer(hello).
No.
```

Disjunctions

Disjunctions:

1. are built using semicolons
2. are true if at least one disjunct is true

Examples:

```
?- ( integer(5); integer(hello); integer(world) ).
Yes.
?- ( integer(hello); integer(world) ).
No.
```

Use parentheses with disjunctions to clarify the structure.

Disjunctions

Special case: multiple answers in case of disjunctions!
⇒ ECL^iPS^e gives separate **Yes** answers for every way in which a disjunctive query can be satisfied.

```
?- ( integer(5) ; integer(7) ).
Yes (0.00s cpu, solution 1, maybe more)
Yes (0.02s cpu, solution 2)
```

```
?- ( integer(5) ; integer(hello) ).
Yes (0.00s cpu, solution 1, maybe more)
No (0.02s cpu)
```

Symbolic Equality

Equality in Prolog:

→ structural equality by pattern matching

- two terms only equal, if they have exactly same structure
- No evaluation of any kind involved

Examples:

```
?- 3 = 3.
Yes.
?- 3 = 4.
No.
?- foo(a,2) = foo(a,2).
Yes.
?- foo(a,2) = foo(b,2).
No.
?- +(3,4) = 7.
No.
?- 3 + 4 = 7.
No.
```

Logical Variables

Logical variables:

1. are variables in the mathematical sense (not in the usual programming language sense)
2. are `placeholders` for values which are not yet known, not labels for storage locations
3. are aliases for logical values and refer to `terms`
4. keep their value once assigned to them
5. are written beginning with an upper-case letter or an underscore, e.g.:
   `X Var Quark _123 R2D2 Wumpus`
6. same identifier multiple times in input term denotes the same variable

Unification

With `logical variables` equality test become much more interesting: **Unification**

Unification:

1. is an extension of pattern matching of two terms
2. also causes `binding` (instantiation, aliasing) of variables in the two terms
3. Idea: instantiate variables such that terms become equal

Examples:

```
X = 7               is true with X instantiated to 7
X = Y               is true with X aliased to Y (or vice versa)
foo(X) = foo(7)     is true with X instantiated to 7
foo(X,Y) = foo(3,4) is true with X instantiated to 3 and Y to 4
foo(X,4) = foo(3,Y) is true with X instantiated to 3 and Y to 4
foo(X) = foo(Y)     is true with X aliased to Y (or vice versa)
foo(X,X) = foo(3,4) is false because no possible value for X
foo(X,4) = foo(3,X) is false because no possible value for X
```

Summary: Basic Terminology

- **Predicate**: Something that is true or false, depending on its definition and its arguments. Defines a relationship between its arguments.
- **Goal**: A logical formula whose truth value we want to know. A goal can be a conjunction or disjunction of other (sub-)goals.
- **Query**: The initial Goal given to a computation.
- **Unification**: An extension of pattern matching which can bind logical variables (placeholders) in the matched terms to make them equal.
- **Clause**: One alternative definition for when a predicate is true. A clause is logically an implication rule.

Programming in Prolog

Comments

Comments can be:

1. **Block comments** enclosed between `/*` and `*/`
2. **Line comments** anything following % in a line (unless '%' character is part of a quoted atom or string)

(Nothing more to say..)

Clauses and facts

In Prolog a `program` is a collection of `predicates`, and a `predicate` is a collection of `clauses`.
So, what is a clause:

1. defines that something is true
2. simplest form is a `fact`, which syntactically is a `structure` or an `atom` terminated by a full stop, e.g.:

```
capital(london, england).
brother(fred, jane).
```

3. General form of a clause:

```
Head :- Body
```

Where `Head` is a `structure` (or `atom`) and `Body` is a `Goal`, e.g.:

```
uncle(X,Z) :- brother(X,Y), parent(Y,Z).
```

Clauses and logical implication

The example clause:

```
uncle(X,Z) :- brother(X,Y), parent(Y,Z).
```

is equivalent to the following **reverse implication**:

$$uncle(X,Z) \leftarrow brother(X,Y) \land parent(Y,Z)$$

or more precisely:

$$\forall X \forall Z : uncle(X,Z) \leftarrow \exists Y : brother(X,Y) \land parent(Y,Z)$$

A `fact` is equivalent to a clause with its `Body` being `true`:

```
brother(fred, jane) :- true.
```

Clauses and predicates

One or multiple clauses with the same head `functor` define(s) a `predicate`, e.g. (with facts):

```
parent(abe, homer).
parent(abe, herbert).
parent(homer, bart).
parent(marge, bart).
```

Logically, multiple clauses:

- are read as disjunctions
- define multiple alternative ways a `predicate` can be `true`

Another example, the `ancestor/2` predicate:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).
```

Execution scheme ⇒ resolution

Resolution:

- **Given**: set of facts and rules as a program
- **Starting point**: a query as an initial goal to be resolved
- **Resolvent**: set of goals that still have to be resolved

Execution scheme ⇒ resolution

Execution mechanism:

1. Pick one goal from the resolvent. If resolvent is empty, stop.
2. Find all clauses whose head successfully unifies with goal. If no such clause, *go to step 6*.
3. Select first of these clauses. If more exist, remember remaining ones. (choice point)
4. Unify goal with head of the selected clause. (may instantiate variables both in the goal and in the clause's body).
5. Prefix this clause body to the resolvent and *go to 1*.
6. Backtrack: Reset whole computation state to how it was when the most recent choice point was created. Take the clauses remembered in this choice point and *go to 3*.

Execution scheme example

```
ancestor(X,Y) :- parent(X,Y).                    % clause 1
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).     % clause 2
parent(abe, homer).                              % clause 3
parent(abe, herbert).                            % clause 4
parent(homer, bart).                             % clause 5
parent(marge, bart).                             % clause 6
```

With query `?- ancestor(X, bart).` we get:

1. both `ancestor/2` predicates can unify the goal
1. but the textually first clause (1) is selected first:

```
Goal (Query):    ancestor(X,bart)
Selected:        clause 1
Unifying:        ancestor(X,bart) = ancestor(X1,Y1)
results in:      X=X1, Y1=bart
New resolvent:   parent(X, bart)
More choices:    clause 2
```

Execution scheme example

```
ancestor(X,Y) :- parent(X,Y).                  % clause 1
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).   % clause 2
parent(abe, homer).                            % clause 3
parent(abe, herbert).                          % clause 4
parent(homer, bart).                           % clause 5
parent(marge, bart).                           % clause 6
```

2. body of clause 1 `parent(X, bart)` is added to resolvent, i.e. a choice point is generated
2. `parent(X, bart)` is next selected for unification
   ⇒ possible matches are clause 5 and 6, try 5 first
2. no body goals to add, the resolvent is now empty

```
Goal:           parent(X, bart)
Selected:       clause 5
Unifying:       parent(X,bart) = parent(homer,bart)
results in:     X = homer
New resolvent:
More choices:   clause 6, then clause 2
```

Execution scheme example

```
ancestor(X,Y) :- parent(X,Y).                % clause 1
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z). % clause 2
parent(abe, homer).                          % clause 3
parent(abe, herbert).                        % clause 4
parent(homer, bart).                         % clause 5
parent(marge, bart).                         % clause 6
```

3. **empty resolvent** ⇒ execution completes successfully, found first solution `X = homer`
3. *ECL^iPS^e* returns solution and asks if more solutions wanted
3. If yes, **backtrack** to most recent *choice point*
3. any **variable bindings** done after *choice point* are **undone**, here binding of `X` to `homer` is undone

```
Goal:           parent(X, bart)
Selected:       clause 6
Unifying:       parent(X,bart) = parent(marge,bart)
results in:     X = marge
New resolvent:
More choices:   clause 2
```

Execution scheme example

```
ancestor(X,Y) :- parent(X,Y).                  % clause 1
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).   % clause 2
parent(abe, homer).                            % clause 3
parent(abe, herbert).                          % clause 4
parent(homer, bart).                           % clause 5
parent(marge, bart).                           % clause 6
```

4. **empty resolvent** ⇒ execution completes successfully, found second solution `X = marge`
4. If still more solutions wanted, **backtrack** to most recent *choice point*
4. no further alternatives for `parent/2` ⇒ check `ancestor/2`

```
Goal:           ancestor(X,bart)
Selected:       clause 2
Unifying:       ancestor(X,bart) = ancestor(X1,Y1)
results in:     Y1 = bart, X1 = X
New resolvent:  parent(Z1, bart), ancestor(X1, Z1)
More choices:
```

Execution scheme example

```
ancestor(X,Y) :- parent(X,Y).                   % clause 1
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).    % clause 2
parent(abe, homer).                             % clause 3
parent(abe, herbert).                           % clause 4
parent(homer, bart).                            % clause 5
parent(marge, bart).                            % clause 6
```

5. new resolvent contains **two goals**: `parent(Z1, bart), ancestor(X1, Z1)`
5. Check leftmost first, `parent(Z1, bart)` ⇒ new *choice point*
5. Select clause 5 first

```
Goal:           parent(Z1, bart)
Selected:       clause 5
Unifying:       parent(Z1, bart) = parent(homer, bart)
results in:     Z1 = homer
New resolvent:  ancestor(X1, homer)
More choices:   clause 6
```

Execution scheme example

```
ancestor(X,Y) :- parent(X,Y).                   % clause 1
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).    % clause 2
parent(abe, homer).                             % clause 3
parent(abe, herbert).                           % clause 4
parent(homer, bart).                            % clause 5
parent(marge, bart).                            % clause 6
```

6. Via finding the ancestor of `homer` a few steps later:

   ```
   ?- ancestor(X,bart).
   X = abe     More? (;)
   ```

6. Finally, `Z1` would be bound to `marge` for whom no ancestors are found ⇒ `False.`

Here, the execution terminates.

More control structures

- Disjunction:

```
at_part(X) :- (X=proton ; X=neutron ; X=electron).
```

  is logically equivalent to:

```
at_part(proton).
at_part(neutron).
at_part(electron).
```

- Conditional:
  - specified by the `->/2` operator
  - combined with `;/2`, a conditional similar to 'if-then-else' can be constructed: `X->Y;Z`
  - only **first solution** of `X` is explored ⇒ no new solutions are tried on **backtracking**!

```
max(X,Y, Max) :-
    number(X), number(Y),
    (X > Y -> Max = X ; Max = Y).
```

More control structures

- Call:
  - in Prolog: data == programs
  - both are represented as terms
  - use predicate `call` to treat terms as goals
  - `call(X)`: at runtime `X` has to be instiantiated, but not at compile time!
  - possible definition of disjunction (`;`):

```
X ; Y :- call(X).
X ; Y :- call(Y).
```

- All solutions:
  - Alternative to one-by-one solution computation:

```
?- findall(X, weekday(X), List).
X = X
List = [mo, tu, we, th, fr, sa, su]
Yes
```

  - See also `setof/3` and `bagof/3` predicates

Using cut: Overview

Use cut (`!`) to prune away part of the Prolog search-space.

- Powerful mechanism to improve program performance
- Suppresses unwanted solutions
- BUT: easily mis- or overused!
- Cut does two things:
  1. **commit**: disregard later clauses for a predicate
  2. **prune**: Throw away alternative solutions to the goal to the left of the cut

Commit to current clause

Consider the following encoding of the “minimum” predicate:

```
min(X,Y, Min) :- X <Y, Min = X.
min(X,Y, Min) :- Y=<X, Min = Y.
```

Problems:

- logically correct, but non-optimal performance
- with `:- min(2,3,M).` Prolog leaves an open *choice point* $\Rightarrow$ during backtracking another minimum would be searched for unnecessarily!
- unnecessary *choice point*:
  1. consumes memory
  2. costs execution time

Solution, use cut:

```
min(X,Y, Min) :- X<Y, !, Min = X.
min(X,Y, Y).
```

Prune alternative solutions

A cut may occur anywhere where a goal may occur:

```
first_prime(X, P) :-
    prime(X,P), !.
```

`first_prime/2`:

- returns the first prime number smaller than `X`
- calls predicate `prime/2`, which generates prime numbers smaller than `X` in descending order
- `!` (cut) prunes away all remaining solutions

→ on backtracking no alternatives are tried

ECL^iPS^e Programming

Structure Notation

Names for structures (so-called declared structures) make them more readable and maintainable, e.g. with:

```
:- local struct( book(author, title, year, publisher) ).
```

Structures with the functor `book/4` can be written as:

```
book{}
book{title:'tom sawyer'}
book{title:'tom sawyer', year:1876, author:twain}
```

which correspond to:

```
book(_, _, _, _)
book(_, 'tom sawyer', _, _)
book(twain, 'tom sawyer', 1876, _)
```

Structure Notation

Properties of declared structure notation:

- the arguments can be written in any order
- “dummy” arguments with anonymous variables do not need to be written
- the arity of the structure is not implied
- the of-syntax can be used to return index of argument, e.g.: `arg(year of book, B, Y)` is equiv. to `arg(3, B, Y)`

Loops

To reduce the need for auxiliary predicates ⇒ iteration construct:

```
( IterationSpecs do Goals )
```

For example, iteration over a list:

```
?- ( foreach(X,[1,2,3]) do writeln(X) )
1
2
3
Yes (0.00s cpu)
```

If a parameter remains **constant** across all loop iterations ⇒ must be specified explicitly (via **param**):

```
?- Array = [](4,3,6,7,8),
   (
        for(I,1,5),
        fromto(0,In,Out,Sum),
        param(Array)
   do
        Out is In + Array[I]
   ).
```

Loops

Possible `IterationSpecs`:

- `fromto(First,In,Out,Last)`: iterate Goals starting with In=First until Out=Last.
- `foreach(X,List)`: iterate Goals with X ranging over all elements of List.
- `foreacharg(X,StructOrArray)`: iterate Goals with X ranging over all arguments of StructOrArray.
- `foreacharg(X,StructOrArray,Idx)`: same as before, but Idx is set to the argument position of X in StructOrArray.
- ...
- `for(I,MinExpr,MaxExpr,[Increment])`: iterate Goals with I ranging over integers from MinExpr to MaxExpr with optional increment.
- ... (see `http://eclipseclp.org/doc/tutorial/tutorial025.html`)

Arrays

Arrays can be of any dimension, indices start at 1, and they are declared with the `dim/2` predicate:

```
?- dim(M,[3,4]).
M = [](([](_131, _132, _133, _134),
       [](_126, _127, _128, _129),
       [](_121, _122, _123, _124))
yes.
```

To query dimensions:

```
?- dim(M,[3,4]), dim(M,D).
...
D = [3, 4]
yes.
```

To access specific elements, specify its index:

```
?- M = []([](2, 3, 5),
          [](1, 4, 7)),  X is M[1, 2] + M[2, 3].
X = 10
M = []([](2, 3, 5), [](1, 4, 7))
yes.
```

Printing

Printing *ECL^{i}PS^e* terms:

- `write(+Stream, ?Term)`: write one term in a default format
- `write_term(+Stream, ?Term, +Options)`: write one term, format options can be selected
- `printf(+Stream, +Format, +ArgList)`: write a string with embedded terms, according to a format string
- `writeq(+Stream, ?Term)`, `write_canonical(+Stream, ?Term)`: write one term so that it can be read back
- `put(+Stream, +Char)`: write one character

Reading

Reading *ECL^{i}PS^e* terms:

- `read(+Stream, -Term, [Options])`: read one fullstop-terminated *ECL^{i}PS^e* term.
- `get(+Stream, -Char)`: read one character
- `read_string(+Stream, +Terminator, -Length, -String)`: read a string up to a certain terminator character
- `read_token(+Stream, -Token, -Class)`: read one syntactic token (e.g. a number, an atom, a bracket, etc)

Example:

```
[eclipse 1]: read(X).
 [3,X,foo(bar),Y].
X = [3, X, foo(bar), Y]
yes.
```

Matching (one-way unification)

Clauses can use **matching** (or one-way unification) instead of head unification:

- written with `?-` functor instead of `:-`
- No variables in the caller will be bound

```
[eclipse 1]: [user].
 p(f(a,X)) ?- writeln(X).
 ?- p(F).
  Query failed: ?- p(F)
 ?- p(f(A,B)).
  Query failed: ?- p(f(A, B))
 ?- p(f(a,b)).
b
```

```
[eclipse 2]: [user].
 p(f(a,X)) :- writeln(X).
 ?- p(f(A,B)).
B
```

List processing

ECL^{i}PS^e provides `append/3`, `length/2`, `member/2`, and `sort/2`:

- `append/3`: append or split lists, e.g.

```
?- append([1, 2], [3, 4], L).
L = [1, 2, 3, 4]
?- append(A, [3, 4], [1, 2, 3, 4]).
A = [1, 2]
?- append([1, 2], B, [1, 2, 3, 4]).
B = [3, 4]
```

- `length/2`: compute the length of a list or construct list of given length, e.g.

```
?- length([1, 2, 3, 4], N).
N = 4
?- length(List, 4).
List = [_1693, _1695, _1697, _1699]
```

List processing

ECL^iPS^e provides `append/3`, `length/2`, `member/2`, and `sort/2`:

- `member/2`: check membership in a list (but memberchk/2 should be preferred), or backtrack over all list members, e.g.

```
?- memberchk(2, [1, 2, 3]).
Yes (0.00s cpu)
?- member(X, [1, 2, 3]).
X = 1
More (0.00s cpu)
X = 2
More (0.01s cpu)
X = 3
Yes (0.01s cpu)
```

- `sort/2`: sort any list and remove duplicates, e.g.

```
?- sort([5, 3, 4, 3, 2], Sorted).
Sorted = [2, 3, 4, 5]
```

Term processing

Generic built-in predicates:

- `=..`: converts structures into lists and vice versa
- `arg/3`: extracts an argument from a structure
- `functor/3`: extracts functor name and arity from structured term
- `term_variables/2`: extracts all variables from arbitrarily complex terms
- `copy_term/2`: creates a copy of a term with fresh variables

Making a module

With `module` directive a new module is declared, e.g.:

```
:- module(greeting).
:- export hello/0.
hello :-
        who(X),
        printf("Hello %w!%n", [X]).
who(world).
who(friend).
```

and with `export` a predicate is exported.

One can now `import` the module and call its exported predicate, e.g.:

```
:- module(main).
:- import greeting. % or ‘import hello/0 from greeting.’
main :-
        hello. % or (without ‘import’) ‘greeting:hello.’
```

Exporting items other than predicates

Most commonly exported items (apart from predicates) are `structure` and `operator` declarations:

```
:- module(data).
:- export struct(employee(name,age,salary)).
:- export op(500, xfx, reports_to).
...
```

Import declarations by importing module (`import data.`).

Interval Constraints Library

Interval Constraints Library

The IC (Interval Constraints) library provides a general interval propagation solver which can be used to solve problems over both integer and real variables.

Load the IC library using either of the following:

```
:- lib(ic).
:- use_module(library(ic)).
```

The typical top-level structure of a constraint program:

```
solve(Variables) :-
        read_data(Data),
        setup_constraints(Data, Variables),
        labeling(Variables).
```

The `labeling/2` predicate is the search part of the program and part of the IC library.

Modeling

Your code must:

- create variables with initial domains
- setup constraints between variables

Example, `SEND+MORE = MONEY`:

```
:- lib(ic).
sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
% Assign finite domain with each letter in list Digits
    Digits :: [0..9],
% Constraints
    alldifferent(Digits),
    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,
% Search
    labeling(Digits).
```

Built-in constraints

The *ic* library provides the following predicates:

- `Vars :: Domain`
  - constrains `Vars` to take only integer or real values from the domain specified by `Domain`
  - `Domain` can be simple range `Lo .. Hi`, or a list of subranges and/or individual elements (integer variables)
  - type of bounds determines type of the variable
  - also allowed: symbolic bound values `inf`, `+inf` and `-inf`
- `Vars $:: Domain`
  - like `::/2`, but for declaring real variables (never imposes integrality, regardless of the types of bounds)
- `Vars #:: Domain`
  - like `::/2`, but for declaring integer variables
- `reals(Vars)` and `integer(Vars)`
  - declares that variables are IC variables / constraints variables to integer values only

Built-in constraints: examples

```
?- X :: -10 .. 10. % integer values from -10 to 10
X = X{-10 .. 10}
Yes
?- X :: -10.0 .. 10.0. % real values from -10 to 10
X = X{-10.0 .. 10.0}
Yes
?- X #:: -10 .. 10. % explicitly declared integer values
X = X{-10 .. 10}
Yes
?- X $:: -10 .. 10. % explicitly declared real values
X = X{-10.0 .. 10.0}
Yes
?- X :: 0 .. 1.0Inf. % integers from zero to infinity
X = X{0 .. 1.0Inf}
Yes
?- X :: 0.0 .. 1.0Inf. % reals from zero to infinity
X = X{0.0 .. 1.0Inf}
Yes
?- X :: [1, 4 .. 6, 9, 10]. % subranges (integers only)
X = X{[1, 4 .. 6, 9, 10]}
Yes
```

Arithmetic constraints in *integral* (#) and *non-integral* ($) version:

- `#=` & `$=`: equality of integer (#) or general ($) expressions
- `#>=` & `$>=`: greater than or equal
- `#=<` & `$=<`: less than or equal
- `#>` & `$>`: greater than
- `#>` & `$>`: less than
- `#\=` & `$\=`: not equal
- `ac_eq(X,Y,C)`: arc-consistent implementation of `X #= Y + C`. `X` and `Y` are constrained to be integer variables and to have "reasonable" bounds. C must be an integer.

Arithmetic constraints

Further notes:

- `X/2 + Y/2 #= 1` and `X + Y #= 2` are **different constraints** ⇒ the first constraints `X` and `Y` to be even
- Except for disequality and `ac_eq/3`, all basic constraints **propagate bound information** (performing interval reasoning), e.g.:

```
?- [X, Y] :: 0 .. 10, X #>= Y + 2.
X = X{2 .. 10}
Y = Y{0 .. 8}
There is 1 delayed goal.
Yes
```

  - `delayed goal` indicates that still some combinations of values for `X` and `Y` violate the constraint
  - constraint remains until no such violation is possible anymore

Constraint Expression Connectives

The following connectives can be used to combine constraint expressions:

1. `and`: Constraint conjunction, e.g. '`X $> 3 and X $< 8`'
2. `or`: Constraint disjunction, e.g. '`X $< 3 or X $> 8`'
3. `=>`: Constraint implication, e.g. '`X $> 3 => X $< 8`'
4. `and`: Constraint negation, e.g. '`neg X $> 3`'

Example:

```
?- [X, Y] :: 0..10, X #>= Y + 6 or X #=< Y - 6.
X = X{0 .. 10}
Y = Y{0 .. 10}
There are 3 delayed goals.
Yes
```

No constraint can be enforced, but …

Constraint Expression Connectives

Once it is known that `X #=< Y - 6` cannot be true, the constraint `X #>= Y +6` is enforced.

Example:

```
?- [X, Y] :: 0..10, X #>= Y + 6 or X #=< Y - 6, X #>= 5.
Y = Y{0 .. 4}
X = X{6 .. 10}
There is 1 delayed goal.
Yes
```

Global constraints

The IC (Interval Constraints) has optional components, which provide *global constraints* ⇒ high-level constraints that provide more global reasoning than the ones in main IC library.

- contained in `ic_global`, `ic_cumulative`, `ic_edge_finder`, and `ic_edge_finder3`
- To use, e.g., `ic_global`:

```
:- lib(ic_global).
:- use_module(library(ic_global)).
```

- Note: some predicates appear in more than one library, e.g., `ic:alldifferent/1` and `ic_global:alldifferent/1`

Different strength of propagation

Compare the `ic:alldifferent/1` version:

```
?- [X1, X2] :: 1 .. 2, [X3, X4] :: 1 .. 4,
   ic:alldifferent([X1, X2, X3, X4]).
X1 = X1{[1, 2]}
X2 = X2{[1, 2]}
X3 = X3{1 .. 4}
X4 = X4{1 .. 4}
There are 4 delayed goals.
Yes
```

with the `ic_global:alldifferent/1` version:

```
?- [X1, X2] :: 1 .. 2, [X3, X4] :: 1 .. 4,
   ic_global:alldifferent([X1, X2, X3, X4]).
X1 = X1{[1, 2]}
X2 = X2{[1, 2]}
X3 = X3{[3, 4]}
X4 = X4{[3, 4]}
There are 2 delayed goals.
Yes
```

Trade-off: longer propagation time for `ic_global` version

Example

N-Queens problem

N-Queens

Place N queens on an $N \times N$ chessboard so that no queen attacks another. A queen attacks all cells in horizontal, vertical and diagonal direction.

- certain solutions for all sizes can be constructed
- not a hard problem
- long history in AI and CP papers

N-Queens problem: Column based Model

- A $1..N$ variable for each column, stating position of queen in the column
- ...because exactly one queen needed for each column
- N variables $\Rightarrow N^2/2$ binary constraints

assign $[X_1, X_2, \ldots, X_N]$ so that:

$$
\begin{aligned}
\forall 1 \leq i \leq N: \quad & X_i && \in 1..N \\
\forall 1 \leq i < j \leq N: \quad & X_i && \neq X_j \\
\forall 1 \leq i < j \leq N: \quad & X_i && \neq X_j + i - j \\
\forall 1 \leq i < j \leq N: \quad & X_i && \neq X_j + j - i
\end{aligned}
$$

Possible program

One possible ECL^iPS^e -program is:

```
:-module(nqueen).
:-export(top/0).
:-lib(ic).
top:-
    nqueen(8,L), writeln(L).
nqueen(N,L):-
    length(L,N),
    L :: 1..N,
    alldifferent(L),
    noattack(L),
    labeling(L).
```

(see: `http://4c.ucc.ie/~hsimonis/ELearning/nqueen/slides.pdf`)

- `noattack/2` defines binary constraints

Binary constraints

Definition of `noattack/2`:

```
noattack([]).
noattack([H|T]):-
    noattack1(H,T,1),
    noattack(T).
noattack1(_,[],_).
noattack1(X,[Y|R],N):-
    X #\= Y+N,
    Y #\= X+N,
    N1 is N+1,
    noattack1(X,R,N1).
```

see: `http://4c.ucc.ie/~hsimonis/ELearning/nqueen/slides.pdf`)

Search improvement

The `search` predicate:

```
[..]
top:-
    nqueen(8,L), writeln(L).
nqueen(N,L):-
    length(L,N),
    L :: 1..N,
    alldifferent(L),
    noattack(L),
    labeling(L). % <-- change this line
```

- last line can be changed to:

```
search(L,0,first_fail,indomain,complete,[])
```

- packaged search library in `ic` constraint solver
- provides many alternative search methods
- just select the right keywords as arguments

search predicate parameters

`search(+L, ++Arg, ++Select, +Choice, ++Method, +Option)`:

1. `L`: List / collection of terms / domain variables
2. `Arg`: 0 if list `L` is list of domain variables, greater than 0 if `L` consists of terms with arity at least Arg
3. `Select`: name of variable selection method (`input_order`, `first_fail`, `smallest`, `largest`, `...`)
4. `Choice`: name of value choice method (`indomain`, `indomain_min`, `indomain_max`, `indomain_middle`, `...`)
5. `Method`: tree search method (`complete`, `bbs(Steps:integer)`, `lds(Disc:integer)`, `...`)
6. `Option`: list of option terms

search improvements

With `first_fail` strategy:

```
search(L,0,first_fail,indomain,complete,[])
```

- solution is different from naive `input_order` approach
- more solutions can be found

Further improvements:

- start from the middle of the board
- use `most_constraint` instead of `first_fail` (tie break based on number of constraints in which variable occurs)
- try **multiple strategies** in parallel (multi-core CPUs)
- limit number of backtracks for search attempt and use randomization (of value and/or variable choice)
- ...

Summary

Summary

ECL^iPS^e :

- interfaces with many programming languages
- provides a stand-alone and a command line user interface
- consists of a collection of libraries
- is intended for problem solving in a declarative fashion enabling rapid prototyping based on the CLP paradigm

Last but not least:

- It is actively developed,
- but already mature enough,
- and very well documented.

We only scratched its surface here.

Check out `http://eclipseclp.org/`, next...

Literature

Sergii Dymchenko & Mariia Mykhailova.
Declaratively solving tricky Google Code Jam problems with Prolog-based ECLiPSe CLP system.
http://arxiv.org/abs/1412.2304

Andrew M. Cheadle, Warwick Harvey, Andrew J. Sadler, Joachim Schimpf, Kish Shen, Mark G. Wallace.
ECLiPSe: A Tutorial Introduction.
http://eclipseclp.org/doc/tutorial/tutorial.html

Abderrahamane Aggoun et al.
ECLiPSe User Manual, Release 6.1.
http://eclipseclp.org/doc/userman/umsroot.html

Helmut Simonis.
ECLiPSE ELearning Website
http://4c.ucc.ie/~hsimonis/ELearning/index.htm