ECL^{i}PS^e Library Manual

Release 4.2

Pascal Brisset, Hani El Sakkout, Thom Frühwirth, Carmen Gervet,
Micha Meier, Stefano Novello, Thierry Le Provost, Joachim Schimpf,
Kish Shen, Mark Wallace

August 6, 1999

Contents

Chapter 1

Introduction

This manual documents the major ECL^{i}PS^e libraries in particular the constaint solver libraries developed at ECRC and IC-Parc. They are enabling tools for the development and delivery of planning and scheduling applications. Since this is an area of active research and new developments, these libaries are subject to technical improvements, addition of new features and redesign as part of our ongoing work. Most of this software is now being developed and maintained in the context of the ICL-sponsored ECL^{i}PS^e-II project, but incorporates contributions from other projects at IC-Parc, in particular the European-funded CHIC-II project.

Chapter 2

The Finite Domains Library

The library **fd.pl** implements constraints over finite domains that can contain integer as well as atomic (i.e. atoms, strings, floats, etc.) and ground compound (e.g. *f(a, b)*) elements. Modules that use the library must start with the directive

:- use_module(library(fd)).

2.1 Terminology

Some of the terms frequently used in this chapter are explained below.

domain variable A domain variable is a variable which can be instantiated only to a value from a given finite set. Unification with a term outside of this domain fails. The domain can be associated with the variable using the predicate **::/2**. Built-in predicates that expect domain variables treat atomic and other ground terms as variables with singleton domains.

integer domain variable An integer domain variable is a domain variable whose domain contains only integer numbers. Only such variables are accepted in inequality constraints and in rational terms. Note that a non-integer domain variable can become an integer domain variable when the non-integer values are removed from its domain.

integer interval An integer interval is written as

$$Min \; .. \; Max$$

with integer expressions $Min \leq Max$ and it represents the set

$$\{\text{Min}, \text{Min} + 1, \ldots, \text{Max}\}.$$

linear term A linear term is a linear integer combination of integer domain variables. The constraint predicates accept linear terms even in a non-canonical form, containing functors +, - and *, e.g.

$$5 * (3 + (4 - 6) * Y - X * 3).$$

If the constraint predicates encounter a variable without a domain, they give it a default domain -10000000..10000000. Note that arithmetic operations on linear terms are performed with single precision integers without any overflow checks. If the domain ranges or coefficients are too large, the operation will not yield correct results. Both the maximum and minimum value of a linear term must be representable as a single precision integer, and so must be the maximum and minimum value of every $c_i x_i$ term.

rational term A rational term is a term constructed from integers and integer domain variables using the arithmetic operations $+, -, *, /$. Besides that, every subexpression of the form *VarA/VarB* must have an integer value in the solution. The system replaces such a subexpression by a new variable *X* and adds a new constraint *VarA #= VarB * X*. Similarly, all subexpressions of the form *VarA*VarB* are replaced by a new variable *X* and a new constraint *X #= VarA * VarB* is added, so that in the internal representation, the term is converted to a linear term.

constraint expression A constraint expression is either an arithmetic constraint or a combination of constraint expressions using the logical FD connectives **#/\/2, #\//2, #=>/2, #<=>/2, #\+/1**.

2.2 Constraint Predicates

?Vars :: ?Domain

Vars is a variable or a list of variables with the associated domain *Domain*. *Domain* can be a closed integer interval denoted as *Min .. Max*, or a list of intervals and/or atomic or ground elements. Although the domain can contain any compound terms that contain no variable, the functor *../2* is reserved to denote integer intervals and thus *1..10* always means an interval and *a..b* is not accepted as a compound domain element.

If *Vars* is already a domain variable, its domain will be updated according to the new domain; if it is instantiated, the predicate checks if the value lies in the domain. Otherwise, if *Vars* is a free variable, it is converted to a domain variable. If *Vars* is a domain variable and *Domain* is free, it is bound to the list of elements and integer intervals representing the domain of the variable (see also **dvar_domain/2** which returns the actual domain).

When a free variable obtains a finite domain or when the domain of a domain variable is updated, the **constrained** list of its **suspend** attribute is woken, if it has any.

::(?Var, ?Domain, ?B)

B is equal to 1 iff the domain of the finite domain variable *Var* is a subset of *Domain* and 0 otherwise.

atmost(+Number, ?List, +Val)

At most *Number* elements of the list *List* of domain variables and ground terms are equal to the ground value *Val*.

constraints_number(+DVar, -Number)

Number is the number of constraints and suspended goals currently attached to the variable *DVar*. Note that this number may not correspond to the exact number of

different constraints attached to *DVar*, as goals in different suspending lists are counted separately. This predicate is often used when looking for the most or least constrained variable from a set of domain variables (see also **deleteffc/3**).

element(?Index, +List, ?Value)

The *Index*'th element of the ground list *List* is equal to *Value*. *Index* and *Value* can be either plain variables, then a domain will be associated to them, or domain variables. Whenever the domain of *Index* or *Value* is updated, the predicate is woken and the domains are updated accordingly.

fd_eval(+E)

The constraint expression *E* is evaluated on runtime and no compile-time processing is performed. This might be necessary in the situations where the default compile-time transformation of the given expression is not suitable, e.g. because it would require type or mode information.

indomain(+DVar)

This predicate instantiates the domain variable *DVar* to elements of its domain, on backtracking the subsequent value is taken. It is used e.g. to find a value of *DVar* which is consistent with all currently imposed constraints. If *DVar* is a ground term, it succeeds. Otherwise, if it is not a domain variable, an error is raised.

is_domain(?Term)

Succeeds if *Term* is a domain variable.

is_integer_domain(?Term)

Succeeds if *Term* is an integer domain variable.

min_max(+Goal, ?C)

If *C* is a linear term, a solution of the goal *Goal* is found that minimises the value of *C*. If *C* is a list of linear terms, the returned solution minimises the maximum value of terms in the list. The solution is found using the *branch and bound* method; as soon as a partial solution is found that is worse than a previously found solution, failure is forced an a new solution is searched for. When a new better solution is found, the bound is updated and the search restarts from the beginning. Each time a new better solution is found, the event 280 is raised. If a solution does not make *C* ground, an error is raised, unless exactly one variable in the list *C* remains free, in which case the system tries to instantiate it to its minimum.

minimize(+Goal, ?Term)

Similar to **min_max/2**, but *Term* must be an integer domain variable. When a new better solution is found, the search does not restart from the beginning, but a failure is forced and the search continues. Each time a new better solution is found, the event 280 is raised. Often **minimize/2** is faster than **min_max/2**, sometimes **min_max/2** might run faster, but it is difficult to predict which one is more appropriate for a given problem.

min_max(+Goal, ?Template, ?Solution, ?C)

minimize(+Goal, ?Template, ?Solution, ?Term)

Similar to **min_max/2** and **minimize/2**, but the variables in *Goal* do not get instantiated to their optimum solutions. Instead, *Solutions* will be unified with a copy of *Template* where the variables are replaced with their minimized values. Typically, the template will contain all or a subset of *Goal*'s variables.

min_max(+Goal, ?C, +Low, +High, +Percent)

minimize(+Goal, ?Term, +Low, +High, +Percent)

Similar to **min_max/2** and **minimize/2**, however the branch and bound method starts with the assumption that the value to be minimised is less than or equal to *High*. Moreover, as soon as a solution is found whose minimised value is less than *Low*, this solution is returned. Solutions within the range of *Percent* % are considered equivalent and so the search for next better solution starts with a minimised value *Percent* % less than the previously found one. *Low*, *High* and *Percent* must be integers.

min_max(+Goal, ?C, +Low, +High, +Percent, +Timeout)

minimize(+Goal, ?Term, +Low, +High, +Percent, +Timeout)

Similar to **min_max/5** and **minimize/5**, but after *Timeout* seconds the search is aborted and the best solution found so far is returned.

min_max(+Goal, ?Template, ?Solution, ?C, +Low, +High, +Percent, +Timeout)

minimize(+Goal, ?Template, ?Solution, ?Term, +Low, +High, +Percent, +Timeout)

The most general variants of the above, with all the optinal parameters.

2.3 Arithmetic Constraint Predicates

?T1 #\= ?T2 The value of the rational term *T1* is not equal to the value of the rational term *T2*.

?T1 #< ?T2 The value of the rational term *T1* is less than the value of the rational term *T2*.

?T1 #<= ?T2 The value of the rational term *T1* is less than or equal to the value of the rational term *T2*.

?T1 #= ?T2 The value of the rational term *T1* is equal to the value of the rational term *T2*.

?T1 #> ?T2 The value of the rational term *T1* is greater than the value of the rational term *T2*.

?T1 #>= ?T2 The value of the rational term *T1* is greater than or equal to the value of the rational term *T2*.

2.4 Logical Constraint Predicates

The logical constraints can be used to combine simpler constraints and to build complex logical constraint expressions. These constraints are preprocessed by the system and transformed into a sequence of evaluation constraints and arithmetic constraints. The logical operators are declared with the following precedences:

```
:- op(750, fy, #\+).
:- op(760, yfx, #/\).
:- op(770, yfx, #\/).
:- op(780, yfx, #=>).
:- op(790, yfx, #<=>).
```

#\+ +E1 *E1* is false, i.e. the logical negation of the constraint expression *E1* is imposed.

+E1 #/\+E2 Both constraint expressions *E1* and *E2* are true. This is equivalent to normal conjunction *(E1, E2)*.

+E1 #\/+E2 At least one of constraint expressions *E1* and *E2* is true. As soon as one of *E1* or *E2* becomes false, the other constraint is imposed.

+E1 #=> +E2 The constraint expression *E1* implies the constraint expression *E2*. If *E1* becomes true, then *E2* is imposed. If *E2* becomes false, then the negation of *E1* will be imposed.

+E1 #<=> +E2 The constraint expression *E1* is equivalent to the constraint expression *E2*. If one expression becomes true, the other one will be imposed. If one expression becomes false, the negation of the other one will be imposed.

2.5 Evaluation Constraint Predicates

These constraint predicates evaluate given constraint expression(s) and associate its truth value with a boolean variable. They can be very useful to define more complex constraints. They can be used both to test entailment of a constraint and to impose a constraint or its negation on the current constraint store.

?B isd +Expr *B* is equal to 1 iff the constraint expression *Expr* is true, 0 otherwise. This predicate is the constraint counterpart of **is/2** - it takes a constraint expression, transforms all its subexpressions into calls to predicates with arity one higher and combines the resulting boolean values to yield *B*. For instance,

B isd X #= Y

is equivalent to

#=(X, Y, B)

#<(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is less than the value of the rational term *T2*, 0 otherwise.

#<=(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is less than or equal to the value of the rational term *T2*, 0 otherwise.

#=(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is equal to the value of the rational term *T2*, 0 otherwise.

#\=(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is different from the value of the rational term *T2*, 0 otherwise.

#>(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is greater than the value of the rational term *T2*, 0 otherwise.

#>=(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is greater than or equal to the value of the rational term *T2*, 0 otherwise.

#/\(+E1, +E2, ?B) *B* is equal to 1 iff both constraint expressions *E1* and *E2* are true, 0 otherwise.

#(+E1, +E2, ?B) *B* is equal to 1 iff at least one of the constraint expressions *E1* and *E2* is true, 0 otherwise.

#<=>(+E1, +E2, ?B) *B* is equal to 1 iff the constraint expression *E1* is equivalent to the constraint expression *E2*, 0 otherwise.

#=>(+E1, +E2, ?B) *B* is equal to 1 iff the constraint expression *E1* implies the constraint expression *E2*, 0 otherwise.

#\+(+E1, ?B) *B* is equal to 1 iff *E1* is false, 0 otherwise.

2.6 CHIP Compatibility Constraints Predicates

These constraints, defined in the module **fd_chip**, are provided for CHIP v.3 compatibility and they are defined using native ECL^{i}PS^e constraints. Their source is available in the file **fd_chip.pl**.

?T1 ## ?T2 The value of the rational term *T1* is not equal to the value of the rational term *T2*.

alldistinct(?List) All elements of *List* (domain variables and ground terms) are pairwise different.

deleteff(?Var, +List, -Rest) This predicate is used to select a variable from a list of domain variables which has the smallest domain. *Var* is the selected variable from *List*, *Rest* is the rest of the list without *Var*.

deleteffc(?Var, +List, -Rest) This predicate is used to select the most constrained variable from a list of domain variables. *Var* is the selected variable from *List* which has the least domain and which has the most constraints attached to it. *Rest* is the rest of the list without *Var*.

deletemin(?Var, +List, -Rest) This predicate is used to select the domain variable with the smallest lower domain bound from a list of domain variables. *Var* is the selected variable from *List*, *Rest* is the rest of the list without *Var*.

List is a list of domain variables or integers. Integers are treated as if they were variables with singleton domains.

dom(+DVar, -List) *List* is the list of elements in the domain of the domain variable *DVar*. The predicate **::/2** can also be used to query the domain of a domain variable, however it yields a list of intervals.

NOTE: This predicate should not be used in ECL^{i}PS^e programs, because all intervals in the domain will be expanded into element lists which causes unnecessary space and time overhead. Unless an explicit list representation is required, finite domains should be processed by the family of the **dom_*** predicates in sections 2.13.2 and 2.13.3.

maxdomain(+DVar, -Max) *Max* is the maximum value in the domain of the integer domain variable *DVar*.

mindomain(+DVar, -Min) *Min* is the minimum value in the domain of the integer domain variable *DVar*.

2.7 Utility Constraints Predicates

These constraints are defined in the module **fd_util** and they consist of useful predicates that are often needed in constraint programs. Their source code is available in the file **fd_util.pl**.

#(?Min, ?CstList, ?Max) The cardinality operator, as described e.g. in [**?**]. *CstList* is a list of constraint expressions and this operator states that at least *Min* and at most *Max* out of them are valid.

dvar_domain_list(?Var, ?List) *List* is the list of elements in the domain of the domain variable or ground term *DVar*. The predicate **::/2** can also be used to query the domain of a domain variable, however it yields a list of intervals.

outof(?Var, +List) The domain variable *Var* is different from all elements of the list *List*.

labeling(+List) The elements of the *List* are instantiated using the **indomain/1** predicate.

2.8 Domain Output

The library **fd_domain.pl** contains output macros which cause an **fd** attribute as well as a domain to be printed as lists that represent the domain values. A domain variable is an attributed variable whose **fd** attribute has a **print** handler which prints it in the same format. For instance,

```
[eclipse 4]: X::1..10, dvar_attribute(X, A), A = fd with domain:D.

X = X{[1..10]}
D = [1..10]
```

```
A = [1..10]
yes.
[eclipse 5]: A::1..10, printf("%mw", A).
A{[1..10]}
A = A{[1..10]}
yes.
```

2.9 Debugging Constraint Programs

The ECL^{i}PS^e debugger is a low-level debugger which is suitable only to debug small constraint programs or to debug small parts of larger programs. Typically, one would use this debugger to debug user-defined constraints and Prolog data processing. When they are known to work properly, this debugger may not be helpful enough to find bugs (correctness debugging) or to speed up a working program (performance debugging). For this, **Grace** (Graphical Constraint Environment) [**?**] is the appropriate tool. Refer to the separate **Grace** documentation for instructions how to use it.

2.10 Debugger Support

The ECL^{i}PS^e debugger supports debugging and tracing of finite domain programs in various ways. First of all, the debugger commands that handle suspended goals can be used to display suspended constraints (**d**, **^**, **u**) or to skip to a particular constraint (**w**, **i**). Note that most of the constraints are displayed using a write macro, their internal form is different.
Successive updates of a domain variable can be traced using the debug event **Hd**. When used, the debugger prompts for a variable name and then it skips to the port at which the domain of this variable was reduced. When a newline is typed instead of a variable name, it skips to the update of the previously entered variable.
A sequence of woken goals can be skipped using the debug event **Hw**.

2.11 Examples

A very simple example of using the finite domains is the *send more money* puzzle:

```
:- use_module(library(fd)).

send(List) :-
    List = [S, E, N, D, M, O, R, Y],
    List :: 0..9,
    alldistinct(List),
    1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E #=
        10000*M+1000*O+100*N+10*E+Y,
    M ## 0,
    labeling(List).
```

The problem is stated very simply, one just writes down the conditions that must hold for the involved variables and then uses the default a *labeling* procedure, i.e. the order in which

the variables will be instantiated. When executing **send/1**, the variables S, M and O are instantiated even before the labeling procedure starts. When a consistent value for the variable E is found (5), and this value is propagated to the other variables, all variables become instantiated and thus the rest of the labeling procedure only checks groundness of the list.
A slightly more elaborate example is the *eight queens* puzzle. Let us show a solution for this problem generalised to N queens and also enhanced by a cost function that evaluates every solution. The cost can be for example $col_i - row_i$ for the i-th queen. We are now looking for the solution with the smallest cost, i.e. one for which the maximum of all $col_i - row_i$ is minimal:

```
:- use_module(library(fd)).

% Find the minimal solution for the N-queens problem
cqueens(N, List) :-
    make_list(N, List),
    List :: 1..N,
    constrain_queens(List),
    make_cost(1, List, C),
    min_max(labeling(List), C).

% Set up the constraints for the queens
constrain_queens([]).
constrain_queens([X|Y]) :-
    safe(X, Y, 1),
    constrain_queens(Y).

safe(_, [], _).
safe(X, [Y|T], K) :-
    noattack(X, Y, K) ,
    K1 is K + 1 ,
    safe(X, T, K1).

% Queens in rows X and Y cannot attack each other
noattack(X, Y, K) :-
    X ## Y,
    X + K ## Y,
    X - K ## Y.

% Create a list with N variables
make_list(0, []) :- !.
make_list(N, [_|Rest]) :-
    N1 is N - 1,
    make_list(N1, Rest).

% Set up the cost expression
make_cost(_, [], []).
make_cost(N, [Var|L], [N-Var|Term]) :-
```

```
        N1 is N + 1,
        make_cost(N1, L, Term).

labeling([]) :- !.
labeling(L) :-
        deleteff(Var, L, Rest),
        indomain(Var),
        labeling(Rest).
```

The approach is similar to the previous example: first we create the domain variables, one for each column of the board, whose values will be the rows. We state constraints which must hold between every pair of queens and finally we make the cost term in the format required for the **min_max/2** predicate. The labeling predicate selects the most constrained variable for instantiation using the **deleteff/3** predicate. When running the example, we get the following result:

```
[eclipse 19]: cqueens(8, X).
Found a solution with cost 5
Found a solution with cost 4

X = [5, 3, 1, 7, 2, 8, 6, 4]
yes.
```

The time needed to find the minimal solution is about five times shorter than the time to generate all solutions. This shows the advantage of the *branch and bound* method. Note also that the board for this 'minimal' solution looks very nice.

2.12 General Guidelines to the Use of Domains

The *send more money* example already shows the general principle of solving problems using finite domain constraints:

- First the variables are defined and their domains are specified.
- Then the constraints are imposed on these variables. In the above example the constraints are simply built-in predicates. For more complicated problems it is often necessary to define Prolog predicates that process the variables are impose constraints on them.
- If stating the constraints alone did not solve the problem, one tries to assign values to the variables. Since every instantiation immediately wakes all constraints associated to the variable, and changes are propagated to the other variables, the search space is usually quickly reduced and either an early failure occurs or the domains of other variables are reduced or directly instantiated. This labeling procedure is therefore incomparably more efficient than the simple *generate and test* algorithm.

The complexity of the program and the efficiency of the solving depends very much on the way these three points are performed. Quite frequently it is possible to state the same problem using different sets of variables with different domains. A guideline is that the search space

should be as small as possible, and thus e.g. five variables with domain 1..10 (i.e. search space size is 10^5) are likely to be better than twenty variables with domain 0..1 (space size 2^{20}).
The choice of constraints is also very important. Sometimes a redundant constraint, i.e. one that follows from the other constraints, can speed up the search considerably. This is because the system does not propagate *all* information it has to all concerned variables, because most of the time this would not bring anything, and thus it would slow down the search. Another reason is that the library performs no meta-level reasoning on constraints themselves (unlike the CHR library). For example, the constraints

```
X + Y #= 10, X + Y + Z #= 14
```

allow only the value 4 for Z, however the system is not able to deduce this and thus it has to be provided as a redundant constraint.
The constraints should be stated in such a way that allows the system to propagate all important domain updates to the appropriate variables.
Another rule of thumb is that creation of choice points should be delayed as long as possible, disjunctive constraints, if there are any, should be postponed as much as possible. Labeling, i.e. value choosing, should be done after all deterministic operations are carried out.
The choice of the labeling procedure is perhaps the most sensitive one. It is quite common that only a very minor change in the order of instantiated variables can speed up or slow down the search by several orders of magnitude. There are only very few common rules available. If the search space is large, it usually pays off to spend more time in selecting the next variable to instantiate. The provided predicates **deleteff/3** and **deleteffc/3** can be used to select the most constrained variable, but in many problems it is possible to extract even more information about which variable to instantiate next.
Often it is necessary to try out several approaches and see how they work, if they do. The profiler and the statistics package can be of a great help here, it can point to predicates which are executed too often, or choice points unnecessarily backtracked over.

2.13 User-Defined Constraints

The **fd.pl** library defines a set of low-level predicates which allow the user to process domain variables and their domains, modify them and write new constraints predicates.

2.13.1 The *fd* Attribute

A domain variable is a metaterm. The **fd.pl** library defines a metaterm attribute

fd with [**domain** : **D**, **min** : **Mi**, **max** : **Ma**, **any** : **A**]

which stores the domain information together with several suspension lists. The attribute arguments have the following meaning:

- **domain** - the representation of the domain itself. Domains are treated as abstract data types, the users should not access them directly, but only using access and modification predicates listed below.
- **min** - a suspension list that should be woken when the minimum of the domain is updated

- **max** - a suspension list that should be woken when the maximum of the domain is updated
- **any** - a suspension list that should be woken when the domain is reduced no matter how.

The suspension list names can be used in the predicate **suspend/3** to denote an appropriate waking condition.
The attribute of a domain variable can be accessed with the predicate **dvar_attribute/2** or by unification in a matching clause:

```
get_attribute(_{fd:Attr}, A) :-
    -?->
    Attr = A.
```

Note however, that this matching clause succeeds even if the first argument is a metaterm but its **fd** attribute is empty. To succeed only for domain variables, the clause must be

```
get_attribute(_{fd:Attr}, A) :-
    -?->
    nonvar(Attr),
    Attr = A.
```

or to access directly attribute arguments, e.g. the domain

```
get_domain(_{fd:fd with domain:D}, Dom) :-
    -?->
    D = Dom.
```

The **dvar_attribute/2** has the advantage that it returns an attribute-like structure even if its argument is already instantiated, which is quite useful when coding **fd** constraints.
The attribute arguments can be accessed by macros from the **structures.pl** library, if e.g. **Attr** is the attribute of a domain variable, the max list can be obtained as

arg(max of fd, Attr, Max)

or, using a unification

Attr = fd with max:Max

2.13.2 Domain Access

The domains are represented as abstract data types, the users are not supposed to access them directly, instead a number of predicates and macros are available to allow operations on domains.

dom_check_in(+Element, +Dom) Succeed if the integer *Element* is in the domain *Dom*.

dom_compare(?Res, +Dom1, +Dom2) Works like **compare/3** for terms. *Res* is unified with

- = iff *Dom1* is equal to *Dom2*,

- $<$ iff *Dom1* is a proper subset of *Dom2*,
- $>$ iff *Dom2* is a proper subset of *Dom1*.

Fails if neither domain is a subset of the other one.

dom_member(?Element, +Dom) Successively instantiate *Element* to the values in the domain *Dom* (similar to **indomain/1**).

dom_range(+Dom, ?Min, ?Max) Return the minimum and maximum value in the integer domain *Dom*. Fails if *Dom* is a domain containing non-integer elements. This predicate can also be used to test if a given domain is integer or not.

dom_size(+Dom, ?Size) *Size* is the number of elements in the domain *Dom*.

2.13.3 Domain Operations

The following predicates operate on domains alone, without modifying domain *variables*. Most of them return the size of the resulting domain which can be used to test if any modification was done.

dom_copy(+Dom1, -Dom2) *Dom2* is a copy of the domain *Dom1*. Since the updates are done in-place, two domain variables must not share the same physical domain and so when defining a new variable with an existing domain, the domain has to be copied first.

dom_difference(+Dom1, +Dom2, -DomDiff, ?Size) The domain *DomDifference* is *Dom1* \ *Dom2* and *Size* is the number of its elements. Fails if *Dom1* is a subset of *Dom2*.

dom_intersection(+Dom1, +Dom2, -DomInt, ?Size) The domain *DomInt* is the intersection of domains *Dom1* and *Dom2* and *Size* is the number of its elements. Fails if the intersection is empty.

dom_union(+Dom1, +Dom2, -DomUnion, ?Size) The domain *DomUnion* is the union of domains *Dom1* and *Dom2* and *Size* is the number of its elements. Note that the main use of the predicate is to yield the most specific generalisation of two domains, in the usual cases the domains become smaller, not bigger.

list_to_dom(+List, -Dom) Convert a list of ground terms and integer intervals into a domain *Dom*. It does not have to be sorted and integers and intervals may overlap.

integer_list_to_dom(+List, -Dom) Similar to **list_to_dom/2** , but the input list should contain only integers and integer intervals and it should be sorted. This predicate will merge adjacent integers and intervals into larges intervals whenever possible. typically, this predicate should be used to convert a sorted list of integers into a finite domain. If the list is known to already contain proper intervals, **sorted_list_to_dom/2** could be used instead.

sorted_list_to_dom(+List, -Dom) Similar to **list_to_dom/2** , but the input list is assumed to be already in the correct format, i.e. sorted and with correct integer and interval values. No checking on the list contents is performed.

2.13.4 Accessing Domain Variables

The following predicates perform various operations:

dvar_attribute(+DVar, -Attrib) *Attrib* is the attribute of the domain variable *DVar*. If *DVar* is instantiated, *Attrib* is bound to an attribute with a singleton domain and empty suspension lists.

dvar_domain(+DVar, -Dom) *Dom* is the domain of the domain variable *DVar*. If *DVar* is instantiated, *Dom* is bound to a singleton domain.

var_fd(+Var, +Dom) If *Var* is a free variable, is becomes a domain variable with the domain *Dom* and with empty suspension lists. The domain *Dom* is copied to make in-place updates logically sound. If *Var* is already a domain variable, its domain is intersected with the domain *Dom*. Fails if *Var* is not a variable.

dvar_msg(+DVar1, +DVar2, -MsgDVar) *MsgVar* is a domain variable which is the most specific generalisation of domain variables or ground values *Var1* and *Var2*.

2.13.5 Modifying Domain Variables

When the domain of a domain variable is reduced, some suspension lists stored in the attribute have to be scheduled and woken.
NOTE: In the **fd.pl** library the suspension lists are woken precisely when the event associated with the list occurs. Thus e.g. the **min** list is woken if and only if the minimum value of the variable's domain is changed, but it is not woken when the variable is instantiated to this minimum or when another element from the domain is removed. In this way, user-defined constraints can rely on the fact that when they are executed, the domain was updated in the expected way. On the other hand, user-defined constraints should also comply with this rule and they should take care not to wake lists when their waking condition did not occur. Most predicates in this section actually do all the work themselves so that the user predicates may ignore scheduling and waking completely.

dvar_remove_element(+DVar, +El) The element *El* is removed from the domain of *DVar* and all concerned lists are woken. If the resulting domain is empty, this predicate fails. If it is a singleton, *DVar* is instantiated. If the domain does not contain the element, no updates are made.

dvar_remove_smaller(+DVar, +El) Remove all elements in the domain of *DVar* which are smaller than the integer *El* and wake all concerned lists. If the resulting domain is empty, this predicate fails, if it is a singleton, *DVar* is instantiated.

dvar_remove_greater(+DVar, +El) Remove all elements in the domain of *DVar* which are greater than the integer *El* and wake all concerned lists. If the resulting domain is empty, this predicate fails, if it is a singleton, *DVar* is instantiated.

dvar_update(+DVar, +NewDom) If the size of the domain *NewDom* is 0, the predicate fails. If it is 1, the domain variable *DVar* is instantiated to the value in the domain. Otherwise, if the size of the new domain is smaller than the size of the domain variable's domain, the domain of *DVar* is replaced by *NewDom*, the appropriate suspension lists

in its attribute are passed to the waking scheduler and so is the **constrained** list in the **suspend** attribute of the domain variable. If the size of the new domain is equal to the old one, no updates and no waking is done, i.e. this predicate does not check an explicit equality of both domains. If the size of the new domain is greater than the old one, an error is raised.

dvar_replace(+DVar, +NewDom) This predicate is similar to **dvar_update/2**, but it does not propagate the changes, i.e. no waking is done. If the size of the new domain is 1, *DVar* is not instantiated, but it is given this singleton domain. This predicate is useful for local consistency checks.

2.14 Extensions

The **fd.pl** library can be used as a basis for further extensions. There are several hooks that make the interfacing easier:

- Each time a new domain variable is created, either in the **::/2** predicate or by giving it a default domain in a rational arithmetic expression, the predicate **new_domain_var/1** is called with the variable as argument. Its default definition does nothing. To use it, it is necessary to redefine it, i.e. to recompile it in the **fd** module, e.g. using **compile/2** or the tool body of **compile_term/1**.

- Default domains are created in the predicate **default_domain/1** in the **fd** module, its default definition is

  ```
  default_domain(Var) :- Var :: -10000000..10000000.
  ```

  It is possible to change default domains by redefining this predicate in the **fd** module.

2.15 Example of Defining a New Constraint

We will demonstrate creation of new constraints on the following example. To show that the constraints are not restricted to linear terms, we can take the constraint

$$X^2 + Y^2 \leq C.$$

Assuming that X and Y are domain variables, we would like to define such a predicate that will be woken as soon as one or both variables' domains are updated in such a way that would require updating the other variable's domain, i.e. updates that would propagate via this constraint. We will define the predicate **sq(X, Y, C)** which will implement this constraint:

```
:- use_module(library(fd)).

% A*A + B*B <= C
sq(A, B, C) :-
    dvar_domain(A, DomA),
    dvar_domain(B, DomB),
    dom_range(DomA, MinA, MaxA),
```

```
dom_range(DomB, MinB, MaxB),
MiA2 is MinA*MinA,
MaB2 is MaxB*MaxB,
(MiA2 + MaB2 > C ->
    NewMaxB is fix(sqrt(C - MiA2)),
    dvar_remove_greater(B, NewMaxB)
;
    NewMaxB = MaxB
),
MaA2 is MaxA*MaxA,
MiB2 is MinB*MinB,
(MaA2 + MiB2 > C ->
    NewMaxA is fix(sqrt(C - MiB2)),
    dvar_remove_greater(A, NewMaxA)
;
    NewMaxA = MaxA
),
(NewMaxA*NewMaxA + NewMaxB*NewMaxB =< C ->
    true
;
    suspend(sq(A, B, C), 3, (A, B)->min)
),
wake.                % Trigger the propagation
```

The steps to be executed when this constraint becomes active, i.e. when the predicate **sq/3** is called or woken are the following:

1. We access the domains of the two variables using the predicate **dvar_domain/2** and and obtain their bounds using **dom_range/3**. Note that it may happen that one of the two variables is already instantiated, but these predicates still work as if the variable had a singleton domain.

2. We check if the maximum of one or the other variable is still consistent with this constraint, i.e. if there is a value in the other variable's domain that satisfies the constraint together with this maximum.

3. If the maximum value is no longer consistent, we compute the new maximum of the domain, and then update the domain so that all values greater than this value are removed using the predicate **dvar_remove_greater/2**. This predicate also wakes all concerned suspension lists and instantiates the variable if its new domain is a singleton.

4. After checking the updates for both variables we test if the constraint is now satisfied for all values in the new domains. If this is not the case, we have to suspend the predicate so that it is woken as soon as the minimum of either domain is changed. This is done using the predicate **suspend/3**.

5. The last action is to trigger the execution of all goals that are waiting for the updates we have made. It is necessary to wake these goals **after** inserting the new suspension, otherwise updates made in the woken goals would not be propagated back to this constraint.

Here is what we get:

```
[eclipse 20]: [X,Y]::1..10, sq(X, Y, 50).

X = X{[1..7]}
Y = Y{[1..7]}

Delayed goals:
sq(X{[1..7]}, Y{[1..7]}, 50)
yes.
[eclipse 21]: [X,Y]::1..10, sq(X, Y, 50), X #> 5.

Y = Y{[1..3]}
X = X{[6, 7]}

Delayed goals:
sq(X{[6, 7]}, Y{[1..3]}, 50)
yes.
[eclipse 22]: [X,Y]::1..10, sq(X, Y, 50), X #> 5, Y #> 1.

X = 6
Y = Y{[2, 3]}
yes.
[eclipse 23]: [X,Y]::1..10, sq(X, Y, 50), X #> 5, Y #> 2.

X = 6
Y = 3
yes.
```

2.16 Program Examples

In this section we present some FD programs that show various aspects of the library usage. If you want to try out the code and experiment with it, the source of most programs is in **demo/extensions/fd**.

2.16.1 Constraining Variable Pairs

The finite domain library gives the user the possibility to impose constraints on the value of a variable. How, in general, is it possible to impose constraints on two or more variables? For example, let us assume that we have a set of colours and we want to define that some colours fit to each other and other do not. This should work in such a way as to propagate possible changes in the domains as soon as this becomes possible.
Let us assume we have a symmetric relation that defines which colours fit to each other:

```
% The basic relation
fit(yellow, blue).
fit(yellow, red).
```

```
fit(blue, yellow).
fit(red, yellow).
fit(green, orange).
fit(orange, green).
```

The predicate **nice_pair(X, Y)** is a constraint and any change of the possible values of X or Y is propagated to the other variable. There are many ways in which this pairing can be defined in ECL^{i}PS^e. They are different solutions with different properties, but they yield the same results.

2.16.1.1 User-Defined Constraints

We use more or less directly the low-level primitives to handle finite domain variables. We collect all consistent values for the two variables, remove all other values from their domains and then suspend the predicate until one of its arguments is updated:

```
nice_pair(A, B) :-
        % get the domains of both variables
    dvar_domain(A, DA),
    dvar_domain(B, DB),
        % make a list of respective matching colours
    setof(Y, X^(dom_member(X, DA), fit(X, Y)), BL),
    setof(X, Y^(dom_member(Y, DB), fit(X, Y)), AL),
        % convert the lists to domains
    sorted_list_to_dom(AL, DA1),
    sorted_list_to_dom(BL, DB1),
        % intersect the lists with the original domains
    dom_intersection(DA, DA1, DA_New, _),
    dom_intersection(DB, DB1, DB_New, _),
        % and impose the result on the variables
    dvar_update(A, DA_New),
    dvar_update(B, DB_New),
        % unless one variable is already instantiated, suspend
        % and wake as soon as any element of the domain is removed
    (var(A), var(B) ->
        suspend(nice_pair3(A, B), 2, [A,B]->any)
    ;
        true
    ).

% Declare the domains
colour(A) :-
    findall(X, fit(X, _), L),
    A :: L.
```

After defining the domains, we can state the constraints:

```
[eclipse 5]: colour([A,B,C]), nice_pair(A, B), nice_pair(B, C), A ## green.
```

```
B = B{[blue, green, red, yellow]}
C = C{[blue, orange, red, yellow]}
A = A{[blue, orange, red, yellow]}

Delayed goals:
  nice_pair(A{[blue, orange, red, yellow]}, B{[blue, green, red, yellow]})
  nice_pair(B{[blue, green, red, yellow]}, C{[blue, orange, red, yellow]})
```

This way of defining new constraints is often the most efficient one, but usually also the most tedious one.

2.16.1.2 Using the *element* Constraint

In this case we use the available primitive in the fd library. Whenever it is necessary to associate a fd variable with some other fd variable, the **element/3** constraint is a likely candidate. Sometimes it is rather awkward to use, because additional variables must be used, but it gives enough power:

```
nice_pair(A, B) :-
    element(I, [yellow, yellow, blue, red, green, orange], A),
    element(I, [blue, red, yellow, yellow, orange, green], B).
```

We define a new variable **I** which is a sort of index into the clauses of the b_nice_pair predicate. The first colour list contains colours in the first argument of b_nice_pair/2 and the second list contains colours from the second argument. The propagation is similar to the previous one. When **element/3** can be used, it is usually faster than the previous approach, because **element/3** is partly hardcoded in C.

2.16.1.3 Using Evaluation Constraints

We can also ancode directly the relations between elements in the domains of the two variables:

```
nice_pair(A, B) :-
    np(A, B),
    np(B, A).

np(A, B) :-
    [A,B] :: [yellow, blue, red, orange, green],
    A #= yellow #=> B :: [blue, red],
    A #= blue #=> B #= yellow,
    A #= red #=> B #= yellow,
    A #= green #=> B #= orange,
    A #= orange #=> B #= green.
```

This method is quite simple and does not need any special analysis, on the other hand it potentially creates a huge number of auxiliary constraints and variables.

2.16.1.4 Using Generalised Propagation

Propia is the first candidate to convert an existing relation into a constraint. One can simply use **infers most** to achieve the propagation:

```
nice_pair(A, B) :-
    fit(A, B) infers most.
```

Using Propia is usually very easy and the programs are short and readable, so that this style of constraints writing is quite useful e.g. for teaching. It is not as efficient as with user-defined constraints, but if the amount of propagation is more important that the efficiency of the constraint itself, it can yield good results, too.

2.16.1.5 Using Constraint Handling Rules

The `domain` solver in CHR can be used directly with the **element/3** constraint as well, however it is also possible to define directly domains consisting of pairs:

```
:- lib(chr).
:- chr(lib(domain)).

nice_pair(A, B) :-
    setof(X-Y, fit(X, Y), L),
    A-B :: L.
```

The pairs are then constrained accordingly:

```
[eclipse 2]: nice_pair(A, B), nice_pair(B, C), A ne orange.

B = B
C = C
A = A

Constraints:
(9) A_g1484 - B_g1516 :: [blue - yellow, green - orange, red - yellow,
yellow - blue, yellow - red]
(10) A_g1484 :: [blue, green, red, yellow]
(12) B_g1516 - C_g3730 :: [blue - yellow, orange - green, red - yellow,
yellow - blue, yellow - red]
(13) B_g1516 :: [blue, orange, red, yellow]
(14) C_g3730 :: [blue, green, red, yellow]
```

2.16.2 Puzzles

Various kinds of puzzles can be easily solved using finite domains. We show here the classical Lewis Carrol's puzzle with five houses and a zebra:

```
Five men with different nationalities live in the first five houses
of a street.  They practise five distinct professions, and each of
```

```
them has a favourite animal and a favourite drink, all of them
different.  The five houses are painted in different colours.

The Englishman lives in a red house.
The Spaniard owns a dog.
The Japanese is a painter.
Thw Italian drinks tea.
The Norwegian lives in the first house on the left.
The owner of the green house drinks coffee.
The green house is on the right of the white one.
The sculptor breeds snails.
The diplomat lives in the yellow house.
Milk is drunk in the middle house.
The Norwegian's house is next to the blue one.
The violinist drinks fruit juice.
The fox is in a house next to that of the doctor.
The horse is in a house next to that of the diplomat.

Who owns a Zebra, and who drinks water?
```

One may be tempted to define five variables Nationality, Profession, Colour, etc. with atomic domains to represent the problem. Then, however, it is quite difficult to express equalities over these different domains. A much simpler solution is to define 5x5 integer variables for each mentioned item, to number the houses from one to five and to represent the fact that e.g. Italian drinks tea by equating Italian = Tea. The value of both variables represents then the number of their house. In this way, no special constraints are needed and the problem is very easily described:

```
:- lib(fd).

zebra([zebra(Zebra), water(Water)]) :-
    Sol = [Nat, Color, Profession, Pet, Drink],
    Nat = [English, Spaniard, Japanese, Italian, Norwegian],
    Color = [Red, Green, White, Yellow, Blue],
    Profession = [Painter, Sculptor, Diplomat, Violinist, Doctor],
    Pet = [Dog, Snails, Fox, Horse, Zebra],
    Drink = [Tea, Coffee, Milk, Juice, Water],

    % we specify the domains and the fact
    % that the values are exclusive
    Nat :: 1..5,
    Color :: 1..5,
    Profession :: 1..5,
    Pet :: 1..5,
    Drink :: 1..5,
    alldifferent(Nat),
    alldifferent(Color),
    alldifferent(Profession),
```

```
    alldifferent(Pet),
    alldifferent(Drink),

    % and here follow the actual constraints
    English = Red,
    Spaniard = Dog,
    Japanese = Painter,
    Italian = Tea,
    Norwegian = 1,
    Green = Coffee,
    Green #= White + 1,
    Sculptor = Snails,
    Diplomat = Yellow,
    Milk = 3,
    Dist1 #= Norwegian - Blue, Dist1 :: [-1, 1],
    Violinist = Juice,
    Dist2 #= Fox - Doctor, Dist2 :: [-1, 1],
    Dist3 #= Horse - Diplomat, Dist3 :: [-1, 1],

    flatten(Sol, List),
    labeling(List).
```

2.16.3 Bin Packing

In this type of problems the goal is to pack a certain amount of different things into the minimal number of bins under specific constraints. Let us solve an example given by Andre Vellino in the Usenet group comp.lang.prolog, June 93:

- There are 5 types of components:

  glass, plastic, steel, wood, copper

- There are three types of bins:

  red, blue, green

- whose capacity constraints are:
  - red has capacity 3
  - blue has capacity 1
  - green has capacity 4
- containment constraints are:
  - red can contain glass, wood, copper
  - blue can contain glass, steel, copper
  - green can contain plastic, wood, copper
- and requirement constraints are (for all bin types):

  wood requires plastic

- Certain component types cannot coexist:
  - glass exclusive copper
  - copper exclusive plastic
- and certain bin types have capacity constraint for certain components
  - red contains at most 1 of wood
  - green contains at most 2 of wood
- Given an initial supply of: 1 of glass, 2 of plastic, 1 of steel, 3 of wood, 2 of copper, what is the minimum total number of bins required to contain the components?

To solve this problem, it is not enough to state constraints on some variables and to start a labeling procedure on them. The variables are namely not known, because we don't know how many bins we should take. One possibility would be to take a large enough number of bins and to try to find a minimum number. However, usually it is better to generate constraints for an increasing fixed number of bins until a solution is found.
The predicate **solve/1** returns the solution for this particular problem, **solve_bin/2** is the general predicate that takes an amount of components packed into a **cont/5** structure and it returns the solution.

```
solve(Bins) :-
    solve_bin(cont(1, 2, 1, 3, 2), Bins).
```

solve_bin/2 computes the sum of all components which is necessary as a limit value for various domains, calls **bins/4** to generate a list **Bins** with an increasing number of elements and finally it labels all variables in the list:

```
solve_bin(Demand, Bins) :-
    Demand = cont(G, P, S, W, C),
    Sum is G + P + S + W + C,
    bins(Demand, Sum, [Sum, Sum, Sum, Sum, Sum, Sum], Bins),
    label(Bins).
```

The predicate to generate a list of bins with appropriate constraints works as follows: first it tries to match the amount of remaining components with zero and the list with nil. If this fails, a new bin represented by a list

[**Colour**, **Glass**, **Plastic**, **Steel**, **Wood**, **Copper**]

is added to the bin list, appropriate constraints are imposed on all the new bin's variables, its contents is subtracted from the remaining number of components, and the predicate calls itself recursively:

```
bins(cont(0, 0, 0, 0, 0), 0, _, []).
bins(cont(G0, P0, S0, W0, C0), Sum0, LastBin, [Bin|Bins]) :-
    Bin = [_Col, G, P, S, W, C],
    bin(Bin, Sum),
    G2 #= G0 - G,
```

```
    P2 #= P0 - P,
    S2 #= S0 - S,
    W2 #= W0 - W,
    C2 #= C0 - C,
    Sum2 #= Sum0 - Sum,
    ordering(Bin, LastBin),
    bins(cont(G2, P2, S2, W2, C2), Sum2, Bin, Bins).
```

The **ordering/2** constraints are strictly necessary because this problem has a huge number of symmetric solutions.
The constraints imposed on a single bin correspond exactly to the problem statement:

```
bin([Col, G, P, S, W, C], Sum) :-
    Col :: [red, blue, green],
    [Capacity, G, P, S, W, C] :: 0..4,
    G + P + S + W + C #= Sum,
    Sum #> 0,               % no empty bins
    Sum #<= Capacity,
    capacity(Col, Capacity),
    contents(Col, G, P, S, W, C),
    requires(W, P),
    exclusive(G, C),
    exclusive(C, P),
    at_most(1, red, Col, W),
    at_most(2, green, Col, W).
```

We will code all of the special constraints with the maximum amount of propagation to show how this can be achieved. In most programs, however, it is not necessary to propagate all values everywhere which simplifies the code quite considerably. Often it is also possible to use some of the built-in symbolic constraints of ECL^{i}PS^e, e.g. **element/3** or **atmost/3**.

2.16.3.1 Capacity Constraints

capacity(Color, Capacity) should instantiate the capacity if the colour is known, and reduce the colour values if the capacity is known to be greater than some values. If we use evaluation constraints, we can code the constraint directly, using equivalences:

```
capacity(Color, Capacity) :-
    Color #= blue #<=> Capacity #= 1,
    Color #= green #<=> Capacity #= 4,
    Color #= red #<=> Capacity #= 3.
```

A more efficient code would take into account the ordering on the capacities. Concretely, if the capacity is greater than 1, the colour cannot be blue and if it is greater than 3, it must be green:

```
capacity(Color, Capacity) :-
    var(Color),
    !,
```

```
        dvar_domain(Capacity, DC),
        dom_range(DC, MinC, _),
        (MinC > 1 ->
            Color ## blue,
            (MinC > 3 ->
                Color = green
            ;
                suspend(capacity(Color, Capacity), 3, (Color, Capacity)->inst)
            )
        ;
            suspend(capacity(Color, Capacity), 3, [Color->inst, Capacity->min])
        ).
    capacity(blue, 1).
    capacity(green, 4).
    capacity(red, 3).
```

Note that when suspended, the predicate waits for colour instantiation or for minimum of the capacity to be updated (except that 3 is one less than the maximum capacity and thus waiting for its instantiation is equivalent).

2.16.3.2 Containment Constraints

The containment constraints are stated as logical expressions and this is also the easiest way to medel them. The important point to remember is that a condition like *red can contain glass, wood, copper* actually means *red cannot contain plastic or steel* which can be written as

```
    contents(Col, G, P, S, W, _) :-
        Col #= red #=> P #= 0 #/\ S #= 0,
        Col #= blue #=> P #= 0 #/\ W #= 0,
        Col #= green #=> G #= 0 #/\ S #= 0.
```

If we want to model the containment with low-level domain predicates, it is easier to state them in the equivalent conjugate form:

- glass can be contained in red or blue
- plastic can be contained in green
- steel can be contained in blue
- wood can be contained in red, green
- copper can be contained in red, blue, green

or in a further equivalent form that uses at most one bin colour:

- glass can not be contained in green
- plastic can be contained in green
- steel can be contained in blue

- wood can not be contained in blue
- copper can be contained in anything

```
contents(Col, G, P, S, W, _) :-
    not_contained_in(Col, G, green),
    contained_in(Col, P, green),
    contained_in(Col, S, blue),
    not_contained_in(Col, W, blue).
```

contained_in(Color, Component, In) states that if Color is different from In, there can be no such component in it, i.e. Component is zero:

```
contained_in(Col, Comp, In) :-
    nonvar(Col),
    !,
    (Col \== In ->
        Comp = 0
    ;
        true
    ).
contained_in(Col, Comp, In) :-
    dvar_domain(Comp, DM),
    dom_range(DM, MinD, _),
    (MinD > 0 ->
        Col = In
    ;
        suspend(contained_in(Col, Comp, In), 1, [Comp->min, Col->inst])
    ).
```

not_contained_in/3 states that if the bin is of the given colour, the component cannot be contained in it:

```
not_contained_in(Col, Comp, In) :-
    nonvar(Col),
    !,
    (Col == In ->
        Comp = 0
    ;
        true
    ).
not_contained_in(Col, Comp, In) :-
    dvar_domain(Comp, DM),
    dom_range(DM, MinD, _),
    (MinD > 0 ->
        Col ## In
    ;
        suspend(not_contained_in(Col, Comp, In), 1, [Comp->min, Col->any])
    ).
```

As you can see again, modeling with the low-level domain predicates might give a faster and more precise programs, but it is much more difficult than using constraint expressions and evaluation constraints. A good approach is thus to start with constraint expressions and only if they are not efficient enough, to (stepwise) recode some or all constraints with the low-level predicates.

2.16.3.3 Requirement Constraints

The constraint 'A requires B' is written as

```
requires(A, B) :-
    A #> 0 #=> B #> 0.
```

With low-level predicates, the constraint 'A requires B' is woken as soon as some A is present or B is known:

```
requires(A, B) :-
    suspend(req(A, B), 1, [A->min, B->inst]).

% woken when A > 0 or B = 0
req(A, B) :-
    (var(B) ->
        B #> 0
    ;
    B = 0 ->
        A = 0
    ;
        true
    ).
```

2.16.3.4 Exclusive Constraints

The exclusive constraint can be written as

```
exclusive(A, B) :-
    A #> 0 #=> B #= 0,
    B #> 0 #=> A #= 0.
```

however a simple form with one disjunction is enough:

```
exclusive(A, B) :-
    A #= 0 #\/ B #= 0.
```

With low-level domain predicates, the exclusive constraint defines a suspension which is woken as soon as one of the two components is present. We also have to take care of the case when both are zero, though:

```
exclusive(A, B) :-
    suspend(excl(A, B), 3, (A,B)->min).
```

```
% woken when A > 0 or B > 0
excl(A, B) :-
    dvar_domain(A, DA),
    dom_range(DA, MinA, _),
    (MinA > 0 ->
        B = 0
    ;           % A did not change its minimum
    B == 0 ->
        true
    ;
        A = 0
    ).
```

2.16.3.5 Atmost Constraints

at_most(N, In, Colour, Components) states that if Colour is equal to In, then there can be at most N Components and vice versa, if there are more than N Components, the colour cannot be In. With constraint expressions, this can be simply coded as

```
at_most(N, In, Col, Comp) :-
    Col #= In #=> Comp #<= N.
```

A low-level solution looks as follows:

```
at_most(N, In, Col, Comp) :-
    nonvar(Col),
    !,
    (In = Col ->
        Comp #<= N
    ;
        true
    ).
at_most(N, In, Col, Comp) :-
    dvar_domain(Comp, DM),
    dom_range(DM, MinM, _),
    (MinM > N ->
        Col ## In
    ;
        suspend(at_most(N, In, Col, Comp), 2, [In->inst, Comp->min])
    ).
```

2.16.3.6 Ordering Constraints

To filter out symmetric solutions we can e.g. impose a lexicographic ordering on the bins in the list, i.e. the second bin must be lexicographically greater or equal than the first one etc. As long as the corresponding most significant variables in two consecutive bins are not instantiated, we cannot constrain the following ones and thus we suspend the ordering on the **inst** lists:

```
ordering([], []).
ordering([Val1|Bin1], [Val2|Bin2]) :-
    Val1 #<= Val2,
    (integer(Val1) ->
        (integer(Val2) ->
            (Val1 = Val2 ->
                ordering(Bin1, Bin2)
            ;
                true
            )
        ;
            suspend(ordering([Val1|Bin1], [Val2|Bin2]), 1, Val2->inst)
        )
    ;
        suspend(ordering([Val1|Bin1], [Val2|Bin2]), 1, Val1->inst)
    ).
```

There is a problem with the representation of the colour: If the colour is represented by an atom, we cannot apply the **#<=/2** predicate on it. To keep the ordering predicate simple and still to have a symbolic representation of the colour in the program, we can define input macros that transform the colour atoms into integers:

```
:- define_macro(no_macro_expansion(blue)/0, tr_col/2, []).
:- define_macro(no_macro_expansion(green)/0, tr_col/2, []).
:- define_macro(no_macro_expansion(red)/0, tr_col/2, []).

tr_col(no_macro_expansion(red), 1).
tr_col(no_macro_expansion(green), 2).
tr_col(no_macro_expansion(blue), 3).
```

2.16.3.7 Labeling

A straightforward labeling would be to flatten the list with the bins and use e.g. **deleteff/3** to label a variable out of it. However, for this very examples not all variables have the same weight – the colour variables propagate much more data when instantiated. Therefore, we first filter out the colours and label them before all the component variables:

```
label(Bins) :-
    colours(Bins, Colors, Things),
    labeleff(Colors),
    flatten(Things, List),
    labeleff(List).

colours([], [], []).
colours([[Col|Rest]|Bins], [Col|Cols], [Rest|Things]) :-
    colours(Bins, Cols, Things).

labeleff([]).
```

```
labeleff(L) :-
    deleteff(V, L, Rest),
    indomain(V),
    labeleff(Rest).
```

Note also that we need a special version of **flatten/3** that works with nonground lists.

2.17 Current Known Restrictions and Bugs

1. The default domain for integer finite domain variables is -10000000..10000000. Larger domains must be stated explicitly using the ::/2 predicate, however neither bound can be outside the long integer range (usually 32 bits).

2. Linear integer terms are processed using long integers and thus if the maximum or minimum value of a linear term overflows this range (usually 32 bits), incorrect results are reported. This may occur if large coefficients are used, if domains are too large or a combination of the two.

Chapter 3

Additional Finite Domain Constraints

3.1 Various Constraints on Lists

The library **fd_global** implements a number of constraints over lists of finite domain variables. It is loaded using one of

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

The following predicates are provided

minlist(+List, ?Min)
Min is the minimum of the values in List. Operationally: Min gets updated to reflect the current range of the minimum of variables and values in List. Likewise, the list elements get constrained to the minimum given.

maxlist(+List, ?Max)
Max is the maximum of the values in List. Operationally: Max gets updated to reflect the current range of the maximum of variables and values in List. Likewise, the list elements get constrained to the maximum given.

ordered(++Relation, +List)
Constrains List to be ordered according to Relation. Relation is one of the atoms <, =<, >, >=, = .

occurrences(++Value, +List, ?N)
The value Value occurs in List N times. Operationally: N gets updated to reflect the number of possible occurrences in the List. List elements may get instantiated to Value, or Value may be removed from their domain if required by N.

sumlist(+List, ?Sum)
The sum of the list elements is Sum. This constraint is more efficient than the general fd-arithmetic constraint if the list is long and Sum is not constrained frequently.

3.2 Cumulative Constraint and Resource Profiles

The library **cumulative** implements the cumulative scheduling constraint. It is based on the finite domain library and is loaded using one of

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

cumulative(+StartTimes, +Durations, +Resources, ++ResourceLimit)
A cumulative scheduling constraint. StartTimes, Durations and Resources are lists of equal length N of finite domain variable or integers. ResourceLimit is an integer. The declarative meaning is: If there are N tasks, each starting at a certain start time, having a certain duration and consuming a certain (constant) amount of resource, then the sum of resource usage of all the tasks does not exceed ResourceLimit at any time.

profile(+StartTimes, +Durations, +Resources, -Profile)
StartTimes, Durations, Resources and Profile are lists of equal length N of finite domain variable or integers with the same meaning as in cumulative/4. The list Profile indicates the level of resource usage at the starting point of each task.

3.3 Edge-finder

The libraries **edge_finder** and **edge_finder3** implement stronger versions of the disjunctive and cumulative scheduling constraints. They employ a technique known as edge-finding to derive stronger bounds on the starting times of the tasks. The library is loaded using either

```
:- use_module(library(edge_finder)).
```

to get a weaker variant with quadratic complexity, or

```
:- use_module(library(edge_finder3)).
```

to get a stronger variant with cubic complexity.

disjunctive(+StartTimes,+Durations)
A disjunctive scheduling constraint. StartTimes and Durations are lists of equal length N of finite domain variable or integers. The declarative meaning is that the N tasks with certain start times and duration do not overlap at any point in time.

cumulative(+StartTimes,+Durations,+Resources,++ResourceLimit)
A cumulative scheduling constraint. StartTimes, Durations and Resources are lists of equal length N of finite domain variable or integers. ResourceLimit is an integer. The declarative meaning is: If there are N tasks, each starting at a certain start time, having a certain duration and consuming a certain (constant) amount of resource, then the sum of resource usage of all the tasks does not exceed ResourceLimit at any time. This constraint can propagate more information than the implementation in library(cumulative).

cumulative(+StartTimes,+Durations,+Resources,+Area,++ResourceLimit)
In this variant, an area (the product of duration and resource usage of a task) can be specified, e.g. if duration or reource usage are not known in advance. The edge-finder algorithm can make use of this information to derive bound updates.

Chapter 4

The Set Domain Library

Conjunto is a system to solve set constraints over finite set domain terms. It has been developed using the kernel of ECL^{i}PS^e based on metaterms. It contains the finite domain library of ECL^{i}PS^e. The library **conjunto.pl** implements constraints over set domain terms that contain herbrand terms as well as ground sets. Modules that use the library must start with the directive

```
:- use_module(library(conjunto))
```

For those who are already familiar with the ECL^{i}PS^e extension manual this manual follows the finite domain library structure.
Note: for any question or information request, please send an email to c.gervet@icparc.ic.ac.uk.

4.1 Terminology

The computation domain of **Conjunto** is finite so set domain and set term will stand respectively for finite set domain and finite set term in the following. Here are defined some of the terms mainly used in the predicate description.

Ground set

A known finite set containing only atoms from the Herbrand Universe or its powerset but without any variable.

Set domain

A discrete lattice or powerset D attached to a set variable S. D is defined by $\{S \in 2^{lub_s} \mid glb_s \subseteq S\}$ under inclusion specified by the notation $Glb_s..Lub_s$. Glb_s and Lub_s represent respectively the intersection and union of elements of D. Thus they are both ground sets. S is then called a **set domain variable**.

Weighted set domain

A specific set domain WD attached to a set variable S where each element of WD is of the form $e(s,w)$. s is a ground set representing a possible value of the set variable and w is the weight or cost associated to this value. e.g.

```
WD = {e(1,50),e({1,3},20)}..{e(1,50),e({1,3},20),e(f(a),100)}.
```

D would have been:

```
{1,{1,3}}..{1,{1,3},f(a)}.
```

Set expression

A composition of set domain variables or ground sets together with set operator symbols which are the standard ones coming from set theory. $S ::= S_1 \cap S_2 \mid S_1 \cup S_2 \mid S_1 \setminus S_2$

Set term

Any term of the followings: (1) a ground set, (2) a set domain variable or (3) a set expression. All set built-in predicates deal with set terms thus with any of the three cases.

4.2 Syntax

- A ground set is written using the characters `{` and `}`, e.g. `S = {1,3,{a,g}, f(2)}`
- A domain D attached to a set variable is specified by two ground sets : $Glb_s..Lub_s$
- Set expressions: Unfortunately the characters representing the usual set operators are not available on our monitors so we use a specific syntax making the connection with arithmetic operators:
  - $\cup$ is represented by \/
  - $\cap$ is represented by /\
  - $\setminus$ is represented by \

4.3 The solver

The **Conjunto** solver acts in a data driven way using a relation between *states*. The transformation performs interval reduction over the set domain bounds. The set expression domains are approximated in terms of the domains of the set variables involved. From a constraint propagation viewpoint this means that constraints over set expressions can be approximated in terms of constraints over set variables. A failure is detected in the constraint propagation phase as soon as one domain lower bound glb_s is not included in its associated upper bound lub_s. Once a solved form has been reached all the constraints which are not definitely solved are delayed and attached to the concerned set variables.

4.4 Constraint predicates

?Svar ‘:: ++Glb..++Lub

attaches a domain to the set variable or to a list of set variables *Svar*. If $Glb \not\subseteq Lub$ it fails. If *Svar* is already a domain variable its domain will be updated according to the new domain; if *Svar* is instantiated it fails. Otherwise if *Svar* is free it becomes a set variable.

set(?Term)

succeeds if *Term* is a ground set.

?S `= ?S1

The value of the set term *S* is equal to the value of the set term *S1*.

?E in ?S

The element *E* is an element of *S*. If *E* is ground it is added to the lower bound of the domain of *S*, otherwise the constraint is delayed. If *E* is ground and does not belong to the upper bound of *S* domain, it fails.

?E notin ?S

The element *E* does not belong to *S*. If *E* is ground it is removed from the upper bound of *S*, otherwise the constraint is delayed. If *E* is ground and belongs to the upper bound of the domain of *S*, it is removed from the upper bound and the constraint is solved. If *E* is ground and belongs to the lower bound of *S* domain, it fails.

?S `< ?S1

The value of the set term *S* is a subset of the value of the set term **S1**. If the two terms are ground sets it just checks the inclusion and succeeds or fails. If the lower bound of the domain of *S* is not included in the upper bound of *S1* domain, it fails. Otherwise it checks the inclusion over the bounds. The constraint is then delayed.

?S `<> ?S1

The domains of S and S1 are disjoint (intersection empty).

all_union(?Lsets, ?S)

Lsets is a list of set variables or ground sets. *S* is a set term which is the union of all these sets. If *S* is a free variable, it becomes a set variable and its attached domain is defined from the union of the domains or ground sets in *Lsets*.

all_disjoint(?Lsets)

Lsets is a list of set variables of ground sets. All the sets are pairwise disjoint.

#(?S,?C)

S is a set term and *C* its cardinality. C can be a free variable, a finite domain variable or an integer. If C is free, this predicate is a mean to access the set cardinality and attach it to C. If not, the cardinality of S is constrained to be C.

sum_weight(?S,?W)

S is a set variable whose domain is a *weighted domain*. *W* is the weight of *S*. If *W* is a free variable, this predicate is a mean to access the set weight and attach it to W. If not, the weight of S is constrained to be W. e.g.

```
S ':: {e(2,3)}..{e(2,3), e(1,4)}, sum_weight(S, W)
```

returns W :: 3..7.

refine(?Svar) and **par_refine(?Svar)**

If *Svar* is a set variable, it labels *Svar* to its first possible domain value. If there are several instances of *Svar*, it creates choice points. If *Svar* is a ground set, nothing happens. Otherwise it fails. **par_refine/1** is the or-parallel variant.

4.5 Examples

4.5.1 Set domains and interval reasoning

First we give a very simple example to demonstrate the expressiveness of set constraints and the propagation mechanism.

```
:- use_module(library(conjunto)).

[eclipse 2]: Car ':: {renault} .. {renault, bmw, mercedes, peugeot},
    Type_french = {renault, peugeot} , Choice '= Car /\ Type_french.

Choice = Choice{{renault} .. {peugeot, renault}}
Car = Car{{renault} .. {bmw, mercedes, peugeot, renault}}
Type_french = {peugeot, renault}

Delayed goals:
      inter_s({peugeot, renault}, Car{{renault}..{bmw, mercedes,
          peugeot, renault}}, Choice{{renault} .. {peugeot, renault}})
yes.
```

If now we add one cardinality constraint:

```
[eclipse 3]: Car ':: {renault} .. {renault, bmw, mercedes, peugeot},
    Type_french = {renault, peugeot} , Choice '= Car /\ Type_french,
    #(Choice, 2).

Car = Car{{peugeot, renault} .. {bmw, mercedes, peugeot, renault}}
Type_french = {peugeot, renault}
Choice = {peugeot, renault}
yes.
```

The first example gives a set of cars from which we know `renault` belongs to. The other labels `{renault, bmw, mercedes, peugeot}` are possible elements of this set. The `Type_french` set is ground and `Choice` is the set term resulting from the intersection of the first two sets. The first execution tells us that `renault` is element of `Choice` and `peugeot` might be one. The intersection constraint is partially satisfied and might be reconsidered if one of the domain of the set terms involved changes. The cosntraint is delayed.

In the second example an additional constraint restricts the cardinality of `Choice` to 2. Satisfying this constraint implies setting the `Choice` set to `{peugeot, renault}`. The domain of this set has been modified so is the intersection constraint activated and solved again. The final result adds `peugeot` to the `Car` set variable. The intersection constraint is now satisfied and removed from the constraint store.

4.5.2 Subset-sum computation with convergent weight

A more elaborate example is a small decision problem. We are given a finite weighted set and a *target* $t \in N$. We ask whether there is a subset s' of S whose weight is t. This also corresponds to having a single weighted set domain and to look for its value such that its weight is t.
This problem is NP-complete. It is approximated in Integer Programming using a procedure which "trims" a list according to a given parameter. For example, the set variable

```
S ':: {}..{e(a,104), e(b,102), e(c,201) ,e(d,101)}
```

is approximated by the set variable

```
S' ':: {}..{e(c,201) ,e(d, 101)}
```

if the parameter delta is 0.04 ($0.04 = 0.2 \div n$ where $n = \#S$).

```
:- use_module(library(conjunto)).

% Find the optimal solution to the subset-sum problem
solve(S1, Sum) :-
        getset(S),
        S1 ':: {}.. S,
        trim(S, S1),
        constrain_weight(S1, Sum),
        sum_weight(S1, W),
        Cost = Sum - W,
        min_max(labeling(S1), Cost).

% The set weight has to be less than Sum
constrain_weight(S1, Sum) :-
        sum_weight(S1, W),
        W #<= Sum.

% Get rid of a set of elements of the set according to a given delta
trim(S, S1) :-
        set2list(S, LS),
        trim1(LS, S1).

trim1(LS, S1) :-
        sort(2, =<, LS, [E | LSorted]),
        getdelta(D),
        testsubsumed(D, E, LSorted, S1).
```

```
testsubsumed(_, _, [], _).
testsubsumed(D, E, [F | LS], S1) :-
        el_weight(E, We),
        el_weight(F, Wf),
        ( We =< (1 - D) * Wf ->
            testsubsumed(D, F, LS, S1)
        ;
            F notin S1,
            testsubsumed(D, E, LS, S1)
        ).

% Instantiation procedure
labeling(Sub) :-
        set(Sub),!.
labeling(Sub) :-
        max_weight(Sub, X),
        ( X in Sub ; X notin Sub ),
        labeling(Sub).

% Some sample data
getset(S) :- S = {e(a,104), e(b,102), e(c,201), e(d,101), e(e,305),
                  e(f,50), e(g,70),e(h,102)}.
getdelta(0.05).
```

The approach is is the following: first create the set domain variable(s), here there is only one which is the set we want to find. We state constraints which limit the weight of the set. We apply the "trim" heuristics which removes possible elements of the set domain. And finally we define the cost term as a finite domain used in the **min_max/2** predicate. The cost term is an integer. The **conjunto.pl** library makes sure that any modification of an fd term involved with a set term is propagated on the set domain. The labeling procedure refines a set domain by selecting the element of the set domain which has the biggest weight using `max_weight(Sub, X)`, and by adding it to the lower bound of the set domain. When running the example, we get the following result:

```
[eclipse 3]: solve(S, 550).
Found a solution with cost 44
Found a solution with cost 24

S = {e(d, 101), e(e, 305), e(f, 50), e(g, 70)}
yes.
```

An interesting point is that in set based problems, the optimization criteria mainly concern the cardinality or the weight of a set term. So in practice we just need to label the set term while applying the **fd** optimization predicates upon the set cardinality or the set weight. There is no need to define additional optimization predicates.

4.5.3 The ternary Steiner system of order n

A ternary Steiner system of order n is a set of $n * (n-1) \setminus 6$ triplets of distinct elements taking their values between *1* and n, such that all the pairs included in two different triplets are different.
This problem is very well dedicated to be solved using set constraints: (i) no order is required in the triplet elements and (ii) the constraint of the problem can be easily written with set constraints saying that any intersection of two set terms contains at most one element. With a finite domain approach, the list of domain variables which should be distinct requires to be given explicitely, thus the problem modelling is would be bit ad-hoc and not valid for any n.

```
:- use_module(library(conjunto)).

% Gives one solution to the ternary steiner problem.
% n has to be congruent to 1 or 3 modulo 6.

steiner(N, LS) :-
        make_nbsets(N,NB),
        make_domain(N, Domain),
        init_sets(NB, Domain, LS),
        card_all(LS, 3),
        labeling(LS, []).

labeling([], _).
labeling([S | LS], L) :-
        refine(S),
        (LS = []  ; LS = [L2 | _Rest],
        all_distincts([S | L], L2),
        labeling(LS, [S | L])).

% the labeled sets are distinct from the set to be labeled
% this constraint is a disjonction so it is useless to put it
% before the labeling as no information would be deduced anyway
all_distincts([], _).
all_distincts([S1 |L], L2) :-
        distinctsfrom(S1, L2),
        all_distincts(L, L2).

distinctsfrom(S, S1) :-
        #(S /\ S1,C),
        C #<= 1.

% creates the required number of set variables according to n
make_nbsets(N,NB) :-
        NB is N * (N-1) // 6.

% initializes the domain of the variables according to n
```

```
make_domain(N, Domain) :-
        D :: 1.. N,
        dom(D, L),
        list2set(L, Domain).

init_sets(0, _Domain, []) :- !.
init_sets(NB, Domain, Sol) :-
        NB1 is NB-1,
        init_sets(NB1, Domain, Sol1),
        S `:: {} .. Domain,
        Sol = [S | Sol1].

% constrains the cardinality of each set variable to be equal to V (=3)
card_all([], _V).
card_all([Set1|LSets], V) :-
        #(Set1, V),
        card_all(LSets, V).
```

The approach with sets is the following: first we create the number of set variables required according to the initial problem definition such that each set variable is a triplet. Then to initialize the domain of these set variables we use the fd predicates which allow to define a domain by an implicit enumeration approach 1..n. This process is cleaner than enumerating a list of integer between 1 and n. Once all the domain variables are created, we constrain their cardinality to be equal to three. Then starts the labeling procedure where all the sets are labeled one after the other. Each time one set is labeled, constraints are stated between the labeled set and the next one to be labeled. This constraint states that two sets have at most one element in common. The semantics of $\#(S \cap S_1, C), C \leq 1$ is equivalent to a disjunction between set values. This implies that in the contraint propagation phase, no information can be deduced until one of the set is ground and some element has been added to the second one. No additional heuristics or tricks have been added to this simple example so it works well for n = 7, 9 but with the value 13 it becomes quite long. When running the example, we get the following result:

```
[eclipse 4]: steiner(7, S).
6 backtracks
0.75
S = [{1, 2, 3}, {1, 4, 5}, {1, 6, 7}, {2, 4, 6}, {2, 5, 7}, {3, 4, 7}, {3, 5, 6}]
yes.
```

4.6 When to use Set Variables and Constraints...

The *subset-sum* example shows that the general principle of solving problems using set domain constraints works just like finite domains:

- Stating the variables and assigning an initial set domain to them.
- Constraining the variables. In the above example the constraint is just a built-in constraint but usually one needs to define additional constraints.

- Labeling the variables, *i.e.*, assigning values to them. In the set case it would not be very efficient to select one value for a set variable for the size of a set domain is exponential in the upper bound cardinality and thus the number of backtracks could be exponential too. A second reason is that no specific information can be deduced from a failure (backtrack) whereas if (like in the refine predicate) we add one by one elements to the set till it becomes ground or some failure is detected, we benefit much more from the constraint propagation mechanism. Every domain modification activates some constraints associated to the variable (depending on the modified bound) and modifications are propagated to the other variables involved in the constraints. The search space is then reduced and either the goal succeeds or it fails. In case of failure the labeling procedure backtracks and removes the last element added to the set variable and tries to instanciate the variable by adding another element to its lower bound. In the `subset-sum` example the labeling only concerns a single set, but it can deal with a list of set terms like in the `steiner` example. Although the choice for the element to be added can be done without specific criterion like in the `steiner` example, some user defined heuristics can be embedded in the labeling procedure like in the `subset-sum` example. Then the user needs to define his own `refine` procedure.

Set constraints propose a new modelling of already solved problems or allows (like for the *subset-sum* example) to solve new problems using CLP. Therefore, one should take into account the problem semantics in order to define the initial search space as small as possible and to make a powerful use of set constraints. The objective of this library is to bring CLP to bear on graph-theorical problems like the *steiner* problem which is a hypergraph computation problem, thus leading to a better specification and solving of problems as, packing and partitioning which find their application in many real life problems. A partial list includes: railroad crew scheduling, truck deliveries, airline crew scheduling, tanker-routing, information retrieval,time tabling problems, location problems, assembly line balancing, political districting,etc.
Sets seem adequate for problems where one is not interested in each element as a specific individual but in a collection of elements where no specific distinction is made and thus where symmetries among the element values need to be avoided (eg. steiner problem). They are also useful when heterogeneous constraints are involved in the problem like weight constraints combined with some disjointness constraints.

4.7 User-defined constraints

To define constraints based on set domains one needs to access the properties of a set term like its domain, its cardinality, its possible weight. As the set variable is a metaterm i.e. an abstract data structure, some built-in predicates allow the user to process the set variables and their domains, modify them and write new constraint predicates.

4.7.1 The abstract set data structure

A set domain variable is a metaterm. The **conjunto.pl** library defines a metaterm attribute **set with [setdom : [Glb,Lub], card: C, weight: W, del_inst: Dinst, del_glb: Dglb, del_lub: Dlub, del_any: Dany]**

This attribute stores information regarding the set domain, its cardinality, and weight (null if undefined) and together with four suspension lists. The attribute arguments have the following meaning:

- **setdom** The representation of the domain itself. As set domains are treated as abstract data types, the users should not access them directly, but only using built-in access and modification predicates presented hereafter.
- **card** The representation of the set cardinality. The cardinality is initialized as soon as a set domain is attached to a set variable. It is either a finite domain or an integer. It can be accessed and modified in the same way as set domains (using specific built-in predicates).
- **weight** The representation of the set weight. The weight is intialized to zero if the domain is not a weighted set domain, otherwise it is computed as soon as a weighted set domain is attached to a set variable. it can be accessed and modified in the same way as set domains (using specific built-in predicates).
- **del_inst** A suspension list that should be woken when the domain is reduced to a single set value.
- **del_glb** A suspension list that should be woken when the lower bound of the set domain is updated.
- **del_lub** a suspension list that should be woken when the upper bound of the set domain is updated.
- **del_any** a suspension list that should be woken when any reduction of the domain is inferred.

The attribute of a set domain variable can be accessed with the predicate **svar_attribute/2** or by unification in a matching clause:

```
get_attribute(_{set: Attr}, A) :- -?-> nonvar(Attr), Attr = A.
```

The attribute arguments can be accessed by macros from the ECL^{i}PS^e**structures.pl** library, if e.g. **Attr** is the attribute of a set domain variable, the del_inst list can be obtained by:

```
arg(del_inst of set, Attr, Dinst)
```

or by using a unification:

```
Attr = set with del_inst: Dinst
```

4.7.2 Set Domain access

The domains are represented as abstract data types, and the users are not supposed to access them directly. So we provide a number of predicates to allow operations on set domains.

set_range(?Svar,?Glb,?Lub)

If *Svar* is a set domain variable, it returns the lower and upper bounds of its domain. Otherwise it fails.

glb(?Svar,?Glb)

If *Svar* is a set domain variable, it returns the lower bound of its domain. Otherwise it fails.

lub(?Svar, ?Lub)

If *Svar* is a set domain variable, it returns the upper bound of its domain. Otherwise it fails.

el_weight(++E, ?We)

If E is element of a weighted domain, it returns the weight associated to E. Otherwise it fails.

max_weight(?Svar,?E)

If *Svar* is a set variable, it returns the element of its domain which belongs to the set resulting from the difference of the upper bound and the lower bound and which has the greatest weight. If *Svar* is a ground set, it returns the element with the biggest weight. Otherwise it fails.

Two specific predicates make a link between a ground set and a list.
set2list(++S, ?L)

If S is a ground set, it returns the corresponding list. If L is also ground it checks if it is the corresponding list. If not, or if S is not ground, it fails.

list2set(++L, ?S)

If L is a ground list, it returns the corresponding set. If S is also ground it checks if it is the corresponding set. If not, or if L is not ground, it fails.

4.7.3 Set variable modification

A specific predicate operate on the set domain *variables*. When a set domain is reduced, some suspension lists have to be scheduled and woken depending on the bound modified.
NOTE: Their are 4 suspension lists in the **conjunto.pl** library, which are woken precisely when the event associated with each list occurs. For example, if the lower bound of a set variable is modified, two suspension lists will be woken: the one associated to a **glb** modification and the one associated to **any** modification. This allows user-defined constraints to be handled efficiently.
modify_bound(Ind, ?S, ++Newbound)

Ind is a flag which should take the value **lub** or **glb**, otherwise it fails ! If S is a ground set, it succeeds if we have *Newbound* equal to S. If S is a set variable, its new lower or upper bound will be updated. For monotonicity reasons, domains can only get reduced. So a new upper bound has to be contained in the old one and a new lower bound has to contain the old one. Otherwise it fails.

4.8 Example of defining a new constraint

The following example demonstrates how to create a new set constraint. To show that set inclusion is not restricted to ground herbrand terms we can take the following constraint which defines lattice inclusion over lattice domains:

```
S_1 incl S
```

Assuming that S and S_1 are specific set variables of the form

```
S `:: {} ..{{a,b,c},{d,e,f}}, ..., S_1 `:: {} ..{{c},{d,f},{g,f}}
```

we would like to define such a predicate that will be woken as soon as one or both set variables' domains are updated in such a way that would require updating the other variable's domain by propagating the constraint. This constraint definition also shows that if one wants to iterate over a ground set (set of known elements) the transformation to a list is convenient. In fact iterations do not suit sets and benefit much more from a list structure. We define the predicate `incl(S,S1)` which corresponds to this constraint:

```
:- use_module(library(conjunto)).
incl(S,S1) :-
          set(S),set(S1),
          !,
          check_incl(S, S1).
incl(S, S1) :-
          set(S),
          set_range(S1, Glb1, Lub1),
          !,
          check_incl(S, Lub1),
          S + Glb1 `= S1NewGlb,
          modify_bound(glb, S1, S1NewGlb).
incl(S, S1) :-
          set(S1),
          set_range(S, Glb, Lub),
          !,
          check_incl(Glb, S1),
          large_inter(S1, Lub, SNewLub),
          modify_bound(lub, S, SNewLub).
incl(S,S1) :-
          set_range(S, Glb, Lub),
          set_range(S1, Glb1, Lub1),
          check_incl(Glb, Lub1),
          Glb \/ Glb1 `= S1NewGlb,
          large_inter(Lub, Lub1, SNewLub),
          modify_bound(glb, S1, S1NewGlb),
          modify_bound(lub, S, SNewLub),
          ( (set(S) ; set(S1)) ->
               true
         ;
```

```
                make_suspension(incl(S, S1),2, Susp),
                insert_suspension([S,S1], Susp, del_any of set, set)
          ),
          wake.

large_inter(Lub, Lub1, NewLub) :-
          set2list(Lub, Llub),
          set2list(Lub1, Llub1),
          largeinter(Llub, Llub1, LNewLub),
          list2set(LNewLub, NewLub).

largeinter([], _, []).
largeinter([S | List_set], Lub1, Snew) :-
          largeinter(List_set, Lub1, Snew1),
          ( contained(S, Lub1) ->
                Snew = [S | Snew1]
          ;
                Snew = Snew1
          ).

check_incl({}, _S) :-!.
check_incl(Glb, Lub1) :-
          set2list(Glb, Lsets),
          set2list(Lub1, Lsets1),
          all_union(Lsets, Union),
          all_union(Lsets1, Union1),
          Union `< Union1,!,
          checkincl(Lsets,Lsets1).
checkincl([], _Lsets1).
checkincl([S | Lsets],Lsets1):-
          contained(S, Lsets1),
          checkincl(Lsets,Lsets1).

contained(_S, []) :- fail,!.
contained(S, [Ss | Lsets1]) :-
          (S `< Ss ->
                true
          ;
                contained(S, Lsets1)
          ).
```

The execution of this constraint is dynamic, *i.e.*, the predicate `incl/2` is called and woken following the following steps:

- We check if the two set variables are ground `set`. If so we just check deterministically if the first one is included (lattice inclusion) in the second one `check_incl`. This predicate checks that any element of a ground set (which is a set itself in this case) is a subset of at least one element of the second set. If not it fails.

- We check if the first set is ground and the second is a set domain variable. If so, `check_incl` is called over the first ground set and the upper bound of the second set. If it succeeds then the lower bound of the set variable might not be consistent any more, we compute the new lower bound (*i.e.*, adding elements from the ground set in it (by using the union predicate) and we modify the bound `modify_bound`. This predicate also wakes all concerned suspension lists and instantiates the set variable if its domain is reduced to a single set (upper bound = lower bound).

- We check if the second set is ground and the first one is a set variable. If so, `check_incl` is called over the lower bound of the first set and the second ground set. If it succeeds then the upper bound of the set variable might not be consistent any more. The new upper bound is computed by intersecting the first set with the upper bound of the set variable in the lattice acceptation `large_inter` and is updated `modify_bound`.

- we check if both set variables are domain variables. If so the lower bound of the first set should be included in the lattice sense in the upper bound of the second one `check/incl`. If it succeeds, then if the lower bound the second set is no more consistent we compute the new one by making the union with first sec lower bound. In the same way, the upper bound of the first set might not be consistent any more. If so, we compute the new one by intersecting (in the lattice acceptation) the both upper bounds to compute the new upper bound of the first set `large_inter`. The upper bound of the first set variable is updated as well as the lower bound of the second set `modify_bound`.

- After checking all these updates, we test if the constraint implies an instanciation of one of the two sets. If this is not the case, we have to suspend the predicate so that it is woken as soon as any bound of either set domain is changed. The predicate `make_suspension/3` can be used for any ECL^{i}PS^e module based on a meta-term structure. It creates a suspension, and then the predicate `insert_suspension/4`, puts this suspension into the appropriate lists (woken when any bound is updated) of both set variables.

- the last action `wake` triggers the execution of all goals that are waiting for the updates we have made. These goals should be woken after inserting the new suspension, otherwise the new updates coming from these woken goals won't be propagated on this constraint !

4.9 Set Domain output

The library **conjunto.pl** contains output macros which print a set variable as well as a ground set respectively as an interval of sets or a set. The **setdom** attribute of a set domain variable (metaterm) is printed in the simplified form of just the *glb..lub* interval, e.g.

```
[eclipse 2]: S ':: {}..{a,v,c}, svar_attribute(S,A), A = set with setdom : D.

S = S{{} .. {a, c, v}}
A = {} .. {a, c, v}
D = [{}, {a, c, v}]
yes.
```

4.10 Debugger

The ECL^{i}PS^e debugger which supports debugging and tracing of finite domain programs in various ways, can just be used the same way for set domain programs. No specific set domain debugger has been implemented for this release.

Chapter 5

Propia - A Library Supporting Generalised Propagation

5.1 Overview

Propia is the name for the implementation of Generalised Propagation in ECL^{i}PS^e.
Generalised propagation is *not* restricted to finite domains, but can be applied to any goal the user cares to specify even if the variables don't have domains.
Effectively the system looks ahead to determine if an approximation to the possible answers has a non-trivial generalization. It is non-trivial if it enables any variables in the goal to become further instantiated, thus reducing search.
The background and motivation for Generalised Propagation is given in references [6, 5, 7]. This section focusses on how to use it. Further examples of the use of Propia are distributed with ECL^{i}PS^e. A simple demonstration of Propia in action on Lewis Carroll's Zebra problem can be run by invoking `lib('propia/zebra')`. An application of Propia to crossword generation can be run by invoking `lib('propia/crossword')`.
Using Propia it is easy to take a standard Prolog program and, with minimal syntactic change, to turn it into a constraint logic program. Any goal `Goal` in the Prolog program, can be transformed into a constraint by annotating it thus `Goal infers most`. The resulting constraint admits just the same answers as the original goal, but its behaviour is quite different. Instead of evaluating the goal by non-deterministically selecting a clause in its definition and evaluating the clause body, Propia evaluates the resulting constraint by extracting information from it deterministically. Propia extracts as much information as possible from the constraints before selecting an ordinary Prolog goal and evaluating it. In this way Propia reduces the number of choices that need to be explored and thus makes programs more efficient.

5.2 Invoking and Using Propia

Propia is an ECL^{i}PS^{e}library, loaded by calling

```
[eclipse]: lib(propia).
```

A goal, such as `member(X,[a,b,c])`, is turned into a constraint by annotating it using the `infers` operator. The second argument of `infers` defines how much propagation should be attempted on the constraint and will be described in section 5.3 below. In this section we

shall use `Goal infers most`, which infers as much information as possible, given the loaded constraint solvers. If the finite domain solver is loaded, then finite domain information is extracted, and Propia reduces the domains to achieve arc-consistency.
We first show the behaviour of the original goal:

```
[eclipse]: member(X,[a,b,c]).

X = a    More? (;)
X = b    More? (;)
X = c    More? (;)
no (more) solution.
```

Constraint propagation is invoked by `infers most`:

```
[eclipse]: lib(fd).
...
[eclipse]: member(X,[a,b,c]) infers most.

X = X{[a,b,c]}
yes.
```

Note that the information produced by the constraint solves the corresponding goal as well. The constraint can thus be dropped.
In case there remains information not yet extracted, the constraint must delay so that completeness is preserved:

```
[eclipse]: member(X,Y) infers most.

X = X
Y = [H3|T3]
Delayed goals:
    member(X, [H3|T3]) infers most
yes.
```

Propia copes correctly with built-in predicates, such as #>and #<:

```
[eclipse]: [user].
 notin3to6(X) :- X#<3.
 notin3to6(X) :- X#>6.
 user compiled
[eclipse]: X::1..10, notin3to6(X) infers most.

X = X{[1, 2, 7 .. 10]}
yes.
```

In this example there are no "delayed" constraints since all valuations for X satisfying the above conditions are solutions. Propia detects this and therefore avoids delaying the constraint again.

In scheduling applications it is necessary to constrain two tasks that require the same machine not to be performed at the same time. Specifically one must end before the other begins, or vice versa. If one task starting at time *ST1* has duration *D1* and another task starting at time *ST2* has duration *D2*, the above "disjunctive" constraint is expressed as follows:

```
[eclipse]: [user].
noclash(ST1,D1,ST2,D2) :- ST1 #>= ST2+D2.
noclash(ST1,D1,ST2,D2) :- ST2 #>= ST1+D1.
 user compiled
```

Generalised Propagation on this constraint allows useful information to be extracted even before it is decided in which order the tasks should be run:

```
[eclipse]: lib(fd).
...
[eclipse]: [ST1,ST2] :: 1..10, noclash(ST1,5,ST2,7) infers most.

ST1 = ST1{[1 .. 5, 8 .. 10]}
ST2 = ST2{[1 .. 3, 6 .. 10]}
Delayed goals:
    noclash(ST1{[1..5, 8..10]}, 5, ST2{[1..3, 6..10]}, 7) infers most
yes.
```

The values *6* and *7* are removed from the domain of *ST1* because the goal `noclash(ST1,5,ST2,7)` cannot be satisfied if *ST1* is either *6* or *7*. For example if *ST1* is *6*, then either $6 > ST2 + 7$ (to satisfy the first clause defining `noclash`) or else $ST2 > 6 + 5$ (to satisfy the second clause). There is no value for $ST2 in \{1...10\}$ that makes either inequality true, and so *6* is removed from the domain of *ST1*. By a similar reasoning *4* and *5* are removed from the domain of *ST2*.
We next take a simple example from propositional logic. In this example the result of constraint propagation is reflected not only in the variable domains, but also in the unification of problem variables. We first define logical conjunction by its truth table:

```
[eclipse]: [user].
 and(true,true,true).
 and(true,false,false).
 and(false,true,false).
 and(false,false,false).
 user  compiled
```

Now we ask for an X, Y, Z satisfying $and(X, Y, Z) \wedge X = Y$. . Both solutions have $X = Y = Z$, and this information is produced solely by propagating on the `and` constraint:

```
[eclipse]: and(X,Y,Z) infers most, X=Y.

Z = X
X = X
Y = X
yes.
```

We now illustrate the potential efficiency benefits of Generalised Propagation with a simple resource allocation problem. A company makes 9 products, each of which require two kinds of components in their manufacture, and yields a certain profit. This information is held in the following table.

```
[eclipse]: [user].
/*** product(Name,#Component1,#Component2,Profit). **/
product(p1,1,19,1).
product(p2,2,17,2).
product(p3,3,15,3).
product(p4,4,13,4).
product(p5,10,8,5).
product(p6,16,4,4).
product(p7,17,3,3).
product(p8,18,2,2).
product(p9,19,1,1).
user compiled
```

We wish to find which products to manufacture in order to make a certain profit without using more than a certain number of either kind of component.[1]
We first define a predicate `sum(Products,Comp1,Comp2,Profit)` which relates a list of products (eg `Products=[p1,p5,p1]`), to the number of each component required to build all the products in the list and the profit (for `[p1,p5,p1]`, `Comp1=12` and `Comp2=46` and `Profit=7`).

```
[eclipse]: user.
sum([],0,0,0).
sum([Name|Products],Count1,Count2,Profit) :-
    [Count1,Count2,Profit]::0..100,
    product(Name,Ct1a,Ct2a,Profita),
    Count1 #= Ct1a+Ct1b,
    Count2 #= Ct2a+Ct2b,
    Profit #= Profita+Profitb,
    sum(Products,Ct1b,Ct2b,Profitb).
user compiled
```

If `sum` is invoked with a list of variables as its first argument, eg `[V1,V2,V3]`, then the only choice made during execution is at the call to `product`. In short, for each variable in the input list there are *9* alternative products that could be chosen. For a list of three variables there are consequently $9^3 = 729$ alternatives.
If we assume a production batch of *9* units, then the number of alternative ways of solving `sum` is 9^9 , or nearly 400 million. To avoid exploring so many possibilities, we simply annotate the call to `product(Name,Ct1a,Ct2a,Profita)` as a Generalised Propagation constraint. Thus the new definition of `sum` is:

```
[eclipse]: user.
sum([],0,0,0).
sum([Name|Products],Count1,Count2,Profit) :-
```

[1] To keep the example simple there is no optimisation.

```
    [Count1,Count2,Profit]::0..100,
    product(Name,Ct1a,Ct2a,Profita) infers most,
    Count1 #= Ct1a+Ct1b,
    Count2 #= Ct2a+Ct2b,
    Profit #= Profita+Profitb,
    sum(Products,Ct1b,Ct2b,Profitb).
user compiled
```

Now `sum` refuses to make any choices:

```
[eclipse]: sum([V1,V2,V3],Comp1,Comp2,Profit).
Comp1 = Comp1{[3..57]}
Comp2 = Comp2{[3..57]}
Profit = Profit{[3..15]}
V3 = V3{[p1, p2, p3, p4, p5, p6, p7, p8, p9]}
V2 = V2{[p1, p2, p3, p4, p5, p6, p7, p8, p9]}
V1 = V1{[p1, p2, p3, p4, p5, p6, p7, p8, p9]}

Delayed goals:
...
```

Using the second version of `sum`, it is simple to write a program which produces lists of products which use less than a given number `Max1` and `Max2` of each component, and yields more than a given profit `MinProfit`:

```
[eclipse]: [user].
solve(Products,Batch,Max1,Max2,MinProfit) :-
    length(Products,Batch),
    Comp1 #<= Max1,
    Comp2 #<= Max2,
    Profit #>= MinProfit,
    sum(Products,Comp1,Comp2,Profit),
    labeling(Products).
user compiled
```

The following query finds which products to manufacture in order to make a profit of 40 without using more than 95 of either kind of component.

```
[eclipse]: solve(P, 9, 95, 95, 40).

P = [p1, p4, p5, p5, p5, p5, p5, p5, p5]     More? (;)
yes.
```

Constraints can be dropped as soon as they became redundant (i.e. as soon as they were entailed by the current partial solution). The check for entailment can be expensive, so Propia only drops constraints if a simple syntactic check allows it. For *infers most*, this check succeeds if the finite domain library is loaded, and the constraint has only one remaining variable.

5.3 Approximate Generalised Propagation

The syntax *Goal infers most* can also be varied to invoke different levels of Generalised Propagation. Other alternatives are *Goal infers fd*, *Goal infers range*, *Goal infers unique*, and *Goal infers consistent*. The strongest constraint is generated by *Goal infers most*, but it can be expensive to compute. The other alternatives may be evaluated more efficiently, and may yield a better overall performance on different applications. We call them "approximations", since the information they produce during propagation is a (weaker) approximation of the information produced by the strongest constraint.
We illustrate the different approximations supported by the current version of Propia on a single small example. The results for *Goal infers most* reflect the problem that structured terms cannot appear in finite domains.

```
[eclipse]: [user].
p(1,a).
p(2,f(Z)).
p(3,3).
user compiled


[eclipse]: p(X,Y) infers most.

X = X{[1..3]}
Y = Y
Delayed goals:
    p(X{[1..3]}, Y) infers most
yes.

[eclipse]: X::[1, 3], p(X, Y) infers most.

X = X{[1, 3]}
Y = Y{[3, a]}
Delayed goals:
    p(X{[1, 3]}, Y{[3, a]}) infers most
yes.

[eclipse]: p(2,Y) infers most.

Y = f(Z)
yes.
```

The first approximation we will introduce in this section is one that searches for the unique answer to the query. It is written *Goal infers unique*. This is cheap because as soon as two different answers to the query have been found, the constraint evaluation terminates and the constraint is delayed again until new information becomes available. Here are two examples of this approximation. In the first example notice that no domain is produced for X.

```
[eclipse]: p(X,Y) infers unique.
```

```
X = X
Y = Y
Delayed goals:
    p(X, Y) infers unique
yes.
```

In the second example, by contrast, `infers unique` yields the same result as `infers most`:

```
[eclipse]: p(X,X) infers unique.
X = 3
yes.
```

The next example shows that *unique* can even capture nonground answers:

```
[eclipse]: p(2,X) infers unique.

X = X
Delayed goals:
    p(2, X) infers unique
yes.
```

The next approximation we shall describe is even weaker: it tests if there is an answer and if not it fails. If there is an answer it checks to see if the constraint is already true.

```
[eclipse]:  p(1,Y) infers consistent.
Y = Y
Delayed goals:
    p(1, Y) infers consistent
yes.

[eclipse]: p(1,a) infers consistent.
yes.

[eclipse]: p(1,X) infers consistent, X=b.
no (more) solution.
```

The strongest language `infers most` extracts any information possible from the loaded constraint solvers. The solvers currently handled by Propia are *unification* (which is the built-in solver of Prolog), *finite domains* and *range*. The finite domain library is loaded by `lib(fd)` and the range library by `lib(range)`. These libraries are described elsewhere. If both libraries are loaded, then `infers most` extracts information from unification, finite domains and ranges. For example:

```
[eclipse]: [user].
 p(f(X),a) :- X *>=0, X *=< 10.
 p(f(X),b) :- X=12.
yes.
[eclipse 14]: p(X,Y) infers most.
```

```
X = f(X{0.0..12.0})
Y = Y{[a, b]}

Delayed goals:
        p(f(X{0.0 .. 12.0}), Y{[a, b]}) infers most
yes.
```

The approximations `infers fd` and `infers range` are similar to `infers most`. However, while `infers most` extracts information based on whatever constraint solvers are loaded, the others only infers information derived from the specified constraint solver. Here's the same example using `infers fd`:

```
[eclipse 14]: p(X,Y) infers fd.

X = f(X)
Y = Y{[a, b]}

Delayed goals:
        p(f(X), Y{[a, b]}) infers fd
yes.
```

Here's the same example using `infers range`:

```
[eclipse 14]: p(X,Y) infers range.

X = f(X{0.0..12.0})
Y = Y

Delayed goals:
        p(f(X{0.0 .. 12.0}), Y) infers range
yes.
```

One rather special approximation langue is `infers ac`, where `ac` stands for arc-consistency. This has similar semantics to `infers fd`, but is implemented very efficiently using the built-in `element` constraint of the finite domain solver. The limitation is that `Goal infers ac` is implemented by executing the goal repeatedly to find all the solutions, and then manipulating the complete set of solutions. It will only work in case there are finitely many solutions and they are all ground.
Finally it is possible to invoke Propia in such a way as to influence its waking conditions. To do this, use the standard `suspend` syntax. For example "forward checking" can be implemented as follows:

```
propagate(Goal,fc) :- !,
    suspend(Goal,4,Goal->inst) infers most.
```

In this case the Propia constraint wakes up each time a variable in the goal is instantiated. The default priority for Propia constraints is 3. However, in the above example, the priority of the Propia constraint has been set to 4.

Chapter 6

The Constraint Handling Rules Library

The `chr` library implements constraint handling rules (CHRs). It includes a compiler, which translates CHR programs into ECL^{i}PS^e programs, and a runtime system. Several constraint handlers and a color graphic demo program are provided in example files in the directory `chr`.
The current `chr` library has now been modified to function correctly without the Opium debugger, which is no longer supported. In addition, the Prolog code produced by the `chr` command is now more readable.
In addition, there is now an experimental extended implementation of CHRs. This extended implementation is faster than the existing `chr` library, and contains some extensions and changes. This is described in section 6.9.

6.1 Introduction

Constraint handling rules (CHRs, CHR home page http://www.pst.informatik.uni-muenchen.de/-fruehwir/chr-intro.html) [2] are a high-level language extension to write *user-defined* constraints. CHRs are essentially a committed-choice language consisting of guarded rules with multiple heads.
The high-level CHRs are an excellent tool for *rapid prototyping* and implementation of constraint handlers. The usual abstract formalism to describe a constraint system, i.e. inference rules, rewrite rules, sequents, formulas expressing axioms and theorems, can be written as CHRs in a straightforward way. Starting from this *executable specification*, the rules can be refined and adapted to the specifics of the application.
CHRs define *simplification* of, and *propagation* over, user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence (e.g. `X>Y,Y>X <=> fail`). Propagation adds new constraints which are logically redundant but may cause further simplification (e.g. `X>Y,Y>Z ==> X>Z`). Repeatedly applying CHRs incrementally simplifies and finally solves user-defined constraints (e.g. `A>B,B>C,C>A` leads to `fail`).
With multiple heads and propagation rules, CHRs provide two features which are essential for non-trivial constraint handling. The declarative reading of CHRs as formulas of first order logic allows one to reason about their correctness. On the other hand, regarding CHRs as a rewrite system on logical formulas allows one to reason about their termination and confluence.

In the next section it is explained how to use CHRs. Then, example constraint handlers and the color graphic demo are listed. The next section introduces the basics of the CHR language and how it works. The next section describes more of the CHR language, the section after the built-in labeling feature. Then there is a section on how to write good CHR programs. Next the debuggers for CHRs are introduced.

6.2 Using Constraint Handling Rules

Here are the steps to be taken from writing to using CHRs:

- Write a CHR program in a file File.`chr`.
- In ECL^{i}PS^e, load the `chr` library with the query `lib(chr)`. It contains both the compiler and runtime system for CHRs. Now ECL^{i}PS^e is in coroutining mode.
- Compile your `chr` file into a `pl` file with the query `chr2pl(File)`.
- In any ECL^{i}PS^e session, you can load a compiled constraint handler (`[File].`). The CHR library is automatically loaded to provide the necessary runtime environment. ECL^{i}PS^e is in coroutining mode.

You can compile your `chr` file and load the resulting `pl` file at once using the query `chr(File)`.

6.3 Example Constraint Handlers

All example files are in the subdirectory `lib/chr` of the installation-directory of ECL^{i}PS^e (which can be found using `get_flag(installation_directory,Dir)`. The files (`.chr, .pl`, examples) relevant to a particular constraint system can be found by looking at all files that match the pattern given in the following listing with each example handler. The examples include a *color graphic demo* about optimal sender placement for wire-less devices in buildings and company sites, small constraint handlers for

- minimum, maximum of and inequalities between terms (`*minmax*`),
- terms (`functor/3, arg/3, =..` as constraints) (`*term*`),
- lists (similar to Prolog III) (`*list*`),
- rational trees (`*tree*`),
- sound if-then-else, negation and checking, lazy conjunction and disjunction (`*control*`),
- geometric reasoning about rectangles (`*demo*`),

and larger constraint handlers for

- booleans for propositional logic (`*bool*`),
- finite and *infinite* domains (inspired by CHIP) (`*domain*`),
- sets (`*set*`),

- terminological reasoning (similar to KL-ONE) [4] (`*kl-one*`),
- temporal reasoning (over time points and intervals) [3] (`*time*`),
- equation solving over real numbers (similar to CLP(R)) or rational numbers (`*math*`).

CHRs have also been used as a committed choice programming language on their own (`*prime*`). The example handlers can be loaded using `chr(lib(File))`. For instance the finite domain handler can be made available as follows (the current directory must have write permission so that the `pl` file can be created):

```
[eclipse 1]: lib(chr), chr(lib(domain)).
...
domain.pl  compiled traceable 241028 bytes in 1.22 seconds

yes.
[eclipse 2]: X::1..10, X ne 5.

X = X

Constraints:
(4) X_g1165 :: [1, 2, 3, 4, 6, 7, 8, 9, 10]

yes.
```

6.4 The CHR Language

User-defined constraints are defined by constraint handling rules - and optional ECL^{i}PS^e clauses for the built-in labeling feature. The constraints must be declared before they are defined. A CHR program (file extension `chr`) may also include other declarations, options and arbitrary ECL^{i}PS^e clauses.

Program ::= Statement [Program]
Statement ::= Declaration | Option | Rule | Clause

Constraint handling rules involving the same constraint can be scattered across a file as long as they are in the same module and compiled together. For readability declarations and options should precede rules and clauses.
In the following subsections, we introduce constraint handling rules and explain how they work. The next section describes declarations, clauses, options and built-in predicates for CHRs.

6.4.1 Constraint Handling Rules

A constraint handling rule has one or two heads, an optional guard, a body and an optional name. A "Head" is a "Constraint". A "Constraint" is an ECL^{i}PS^e *callable term* (i.e. atom or structure) whose functor is a declared constraint. A "Guard" is an ECL^{i}PS^e goal. The *guard is a test* on the applicability of a rule. The "Body" of a rule is an ECL^{i}PS^e goal (including constraints). The execution of the guard and the body should not involve side-effects (like

`assert/1`, `write/1`) (for more information see the section on writing CHR programs). A rule can be named with a "RuleName" which can be any ECL^{i}PS^e term (including variables from the rule). During debugging (see section 6.8), this name will be displayed instead of the whole rule.
There are three kinds of constraint handling rules.

Rule	::= SimplificationRule | PropagationRule | SimpagationRule
SimplificationRule	::= [RuleName `@`] Head [`,` Head] `<=>` [Guard `|`] Body.
PropagationRule	::= [RuleName `@`] Head [`,` Head] `==>` [Guard `|`] Body.
SimpagationRule	::= [RuleName `@`] Head `\` Head `<=>` [Guard `|`] Body.

Declaratively, a rule relates heads and body *provided the guard is true*. A simplification rule means that the heads are true if and only if the body is true. A propagation rule means that the body is true if the heads are true. A simpagation rule is a combination of a simplification and propagation rule. The rule "Head1 `\` Head2 `<=>` Body" is equivalent to the simplification rule "Head1 `,` Head2 `<=>` Body`,` Head1." However, the simpagation rule is more compact to write, more efficient to execute and has better termination behavior than the corresponding simplification rule.
Example: Assume you want to write a constraint handler for minimum and maximum based on inequality constraints. The complete code can be found in the handler file `minmax.chr`.

```
 handler minmax.

constraints leq/2, neq/2, minimum/3, maximum/3.
built_in     @ X leq Y <=> \+nonground(X),\+nonground(Y) | X @=< Y.
reflexivity  @ X leq X <=> true.
antisymmetry @ X leq Y, Y leq X <=> X = Y.
transitivity @ X leq Y, Y leq Z ==> X \== Y, Y \== Z, X \== Z | X leq Z.
...
built_in     @ X neq Y <=> X \== Y | true.
irreflexivity@ X neq X <=> fail.
...
subsumption  @ X lss Y \ X neq Y <=> true.
simplification @ X neq Y, X leq Y <=> X lss Y.
...
min_eq @ minimum(X, X, Y) <=> X = Y.
min_eq @ minimum(X, Y, X) <=> X leq Y.
min_eq @ minimum(X, Y, Y) <=> Y leq X.
...
propagation @ minimum(X, Y, Z) ==> Z leq X, Z leq Y.
...
```

Procedurally, a rule can fire only if its guard succeeds. A firing simplification rule *replaces* the head constraints by the body constraints, a firing propagation rule keeps the head constraints and *adds* the body. A firing simpagation rule keeps the first head and replaces the second head by the body. See the next subsection for more details.

6.4.2 How CHRs Work

ECL^{i}PS^e will first solve the built-in constraints, then user-defined constraints by CHRs then the other goals.
Example, contd.:

```
[eclipse]: chr(minmax).
minmax.chr compiled traceable 106874 bytes in 3.37 seconds
minmax.pl  compiled traceable 124980 bytes in 1.83 seconds
yes.
[eclipse]: minimum(X,Y,Z), maximum(X,Y,Z).
X = Y = Z = _g496
yes.
```

Each user-defined constraint is associated with all rules in whose heads it occurs by the CHR compiler. Every time a user-defined constraint goal is added or re-activated, it checks itself the applicability of its associated CHRs by *trying* each CHR. To try a CHR, one of its heads is matched against the constraint goal. If a CHR has two heads, the constraint store is searched for a "partner" constraint that matches the other head. If the matching succeeded, the guard is executed as a test. Otherwise the rule delays and the next rule is tried.
The guard either succeeds, fails or delays. If the guard succeeds, the rule fires. Otherwise the rule delays and the next rule is tried. In the current implementation, a guard succeeds if its execution succeeds without delayed goals and attempts to "touch" a global variable (one that occurs in the heads). A variable is *touched* if it is unified with a term (including other variables), if it gets more constrained by built-in constraints (e.g. finite domains or equations over rationals) or if a goal delays on it (see also the `check_guard_bindings` option). Currently, built-in constraints used in a guard act as tests only (see also the section on writing good CHR programs).
If the firing CHR is a simplification rule, the matched constraint goals are removed and the body of the CHR is executed. Similarly for a firing simpagation rule, except that the first head is kept. If the firing CHR is a propagation rule the body of the CHR is executed and the next rule is tried. It is remembered that the propagation rule fired, so it will not fire again (with the same partner constraint) if the constraint goal is re-activated.
If the constraint goal has not been removed and all rules have been tried, it delays until a variable occurring in the constraint is touched. Then the constraint is re-activated and all its rules are tried again.
Example, contd.: The following trace is edited, rules that are tried in vain and redelay have been removed.

```
[eclipse]: chr_trace.
yes.
Debugger switched on - creep mode
[eclipse]: notrace.     % trace only constraints
Debugger switched off
yes.
[eclipse]: minimum(X,Y,Z), maximum(X,Y,Z).

ADD (1) minimum(X, Y, Z)
```

```
TRY (1) minimum(_g218, _g220, _g222) with propagation
RULE 'propagation' FIRED

 ADD (2) leq(_g665, _g601)

 ADD (3) leq(_g665, Var)

ADD (4) maximum(_g601, Var, _g665)
TRY (4) maximum(_g601, Var, _g665) with propagation
RULE 'propagation' FIRED

 ADD (5) leq(_g601, _g665)
 TRY (5) leq(_g601, _g665) (2) leq(_g665, _g601) with antisymmetry
 RULE 'antisymmetry' FIRED

TRY (4) maximum(_g601, Var, _g601) with max_eq
RULE 'max_eq' FIRED

 ADD (6) leq(Var, _g601)
 TRY (3) leq(_g601, Var) (6) leq(Var, _g601) with antisymmetry
 RULE 'antisymmetry' FIRED

TRY (1) minimum(_g601, _g601, _g601) with min_eq
RULE 'min_eq' FIRED

 ADD (7) leq(_g601, _g601)
 TRY (7) leq(_g601, _g601) with reflexivity
 RULE 'reflexivity' FIRED

X = Y = Z = _g558
yes.
```

6.5 More on the CHR Language

The following subsections describe declarations, clauses, options and built-in predicates of the CHR language.

6.5.1 Declarations

Declarations name the constraint handler, its constraints, specify their syntax and use in built-in labeling.

Declaration	::= `handler` Name.
	::= `constraints` SpecList.
	::= `operator`(Precedence,Associativity,Name).
	::= `label_with` Constraint `if` Guard.

The optional `handler` declaration documents the name of the constraint handler. Currently it can be omitted, but will be useful in future releases for combining handlers.
The mandatory `constraints` declaration lists the constraints defined in the handler. A "SpecList" is a list of Name/Arity pairs for the constraints. The declaration of a constraint *must appear before* the constraint handling rules and ECL^{i}PS^e clauses which define it, otherwise a syntax error is raised. There can be several `constraints` declarations.
The optional `operator` declaration declares an operator, with the same arguments as `op/3` in ECL^{i}PS^e. However, while the usual operator declarations are ignored during compilation from `chr` to `pl` files, the CHR operator declarations are taken into account (see also the subsection on clauses).
The optional `label_with` declaration specifies when the ECL^{i}PS^e clauses of a constraint can be used for built-in labeling (see subsection on labeling).
Example, contd.: The first lines of the minmax handler are declarations:

```
handler minmax.

constraints leq/2, neq/2, minimum/3, maximum/3.

operator(700, xfx, leq).
operator(700, xfx, neq).
```

6.5.2 ECL^{i}PS^e Clauses

A constraint handler program may also include arbitrary ECL^{i}PS^e code (written with the four operators `:- /[1,2]` and `?- /[1,2]`).

Clause ::= Head `:-` Body.
::= Head `?-` Body.
::= `:-` Body.
::= `?-` Body.

Note that `:-/1` and `?-/1` *behave different from each other* in CHR programs. Clauses starting with `:-` are *copied* into the `pl` file by the CHR compiler, clauses with `?-` are *executed* by the compiler. As the `op` declaration needs both copying and execution, we have introduced the special `operator` declaration (see previous subsection on declarations). A "Head" can be a "Constraint", such clauses are used for built-in labeling only (see section on labeling).

6.5.3 Options

The `option` command allows the user to set options in the CHR compiler.

Option ::= `option(Option, On_or_off).`

Options can be switched on or off. *Default is* `on`. Advanced users may switch an option off to improve the efficiency of the handler at the cost of safety. Options are:

- `check_guard_bindings`: When executing a guard, it is checked that no global variables (variables of the rule heads) are touched (see subsection on how CHRs work). If the option is on, guards involving cut, if-then-else or negation may not work correctly if

a global variable has been touched before. If switched off, guard checking may be significantly faster, but only safe if the user makes sure that global variables are not touched. To ensure that the variables are sufficiently bound, tests like `nonvar/1` or delays can be added to the predicates used in the guards.

- `already_in_store`: Before adding a user-defined constraint to the constraint store, it is checked if there is an identical one already in the store. If there is, the new constraint needs not to be added. The handling of the duplicate constraint is avoided. This option can be set to `off`, because the checking may be too expensive if duplicate constraints rarely occur. Specific duplicate constraints can still be removed by a simpagation rule of the form `Constraint \ Constraint <=> true`.

- `already_in_heads`: In two-headed simplification rules, the intention is often to simplify the two head constraints into a stronger version of one of the constraints. However, a straightforward encoding of the rule may include the case where the new constraint is identical to the corresponding head constraint. Removing the head constraint and adding it again in the body is inefficient and may cause termination problems. If the `already_in_heads` option is on, in such a case the head constraint is kept and the body constraint ignored. Note however, that this optimization currently *only works if* the body constraint is the only goal of the body or the first goal in the conjunction comprising the body of the rule (see the example handler for domains). The option may be too expensive if identical head-body constraints rarely occur.

- Note that the ECL^{i}PS^e environment flag `debug_compile` (set and unset with `dbgcomp` and `nodbgcomp`) is also taken into account by the CHR compiler. The default is `on`. If switched off, the resulting code is more efficient, but cannot be debugged anymore (see section 6.8).

6.5.4 CHR Built-In Predicates

There are some built-in predicates to compile `chr` files, for debugging, built-in labeling and to inspect the constraint store and remove its constraints:

- `chr2pl`(File) compiles "File" from a `chr` to `pl` file.

- `chr`(File) compiles "File" from a `chr` to `pl` file and loads the `pl` file.

- `chr_trace` activates the standard debugger and shows constraint handling.

- `chr_notrace` stops either debugger.

- `chr_labeling` provides built-in labeling (see corresponding subsection).

- `chr_label_with`(Constraint) checks if "Constraint" satisfies a `label_with` declaration (used for built-in labeling).

- `chr_resolve`(Constraint) uses the ECL^{i}PS^e clauses to solve a constraint (used for built-in labeling).

- `chr_get_constraint`(Constraint) gets a constraint unifying with "Constraint" from the constraint store and removes it, gets another constraint on backtracking.

- `chr_get_constraint(Variable,Constraint)` is the same as `chr_get_constraint/1` except that the constraint constrains the variable "Variable".

6.6 Labeling

In a constraint logic program, constraint handling is interleaved with making choices. Typically, without making choices, constraint problems cannot be solved completely. *Labeling* is a controlled way to make choices. Usually, a labeling predicate is called at the end of the program which chooses values for the variables constrained in the program. We will understand labeling in the most general sense as a procedure introducing arbitrary choices (additional constraints on constrained variables) in a systematic way.
The CHR run-time system provides *built-in labeling* for user-defined constraints. The idea is to write clauses for user-defined constraints that are used for labeling the variables in the constraint. These clauses are not used during constraint handling, but only during built-in labeling. Therefore the "Head" of a clause may be a user-defined "Constraint". The `label_with` declaration restricts the use of the clauses for built-in labeling (see subsection on declarations). There can be several `label_with` declarations for a constraint.
Example, contd.:

```
label_with minimum(X, Y, Z) if true.
minimum(X, Y, Z):- X leq Y, Z = X.
minimum(X, Y, Z):- Y lss X, Z = Y.
```

The built-in labeling is invoked by calling the CHR built-in predicate `chr_labeling/0` (no arguments). Once called, whenever no more constraint handling is possible, the built-in labeling will choose a constraint goal whose `label_with` declaration is satisfied for labeling. It will introduce choices using the clauses of the constraint.
Example, contd.: A query without and with built-in labeling:

```
[eclipse]: minimum(X,Y,Z), maximum(X,Y,W), Z neq W.

X = _g357
Y = _g389
Z = _g421
W = _g1227

Constraints:
(1) minimum(_g357, _g389, _g421)
(2) _g421 leq _g357
(3) _g421 leq _g389
(4) maximum(_g357, _g389, _g1227)
(5) _g357 leq _g1227
(7) _g389 leq _g1227
(10) _g421 lss _g1227

yes.

[eclipse]: minimum(X,Y,Z), maximum(X,Y,W), Z neq W, chr_labeling.
```

```
X = Z = _g363
Y = W = _g395

Constraints:
(10) _g363 lss _g395

     More? (;)

X = W = _g363
Y = Z = _g395

Constraints:
(17) _g395 lss _g363

yes.
```

Advanced users can write their own labeling procedure taking into account the constraints in the constraint store (see next subsection for CHR built-in predicates to inspect and manipulate the constraint store).
Example The predicate `chr_labeling/0` can be defined as:

```
labeling :-
        chr_get_constraint(C),
        chr_label_with(C),
        !,
        chr_resolve(C),
        labeling.
labeling.
```

6.7 Writing Good CHR Programs

This section gives some programming hints. For maximum efficiency of your constraint handler, see also the subsection on options, especially on `check_guard_bindings` and the `debug_compile` flag.

6.7.1 Choosing CHRs

Constraint handling rules for a given constraint system can often be derived from its definition in formalisms such as inference rules, rewrite rules, sequents, formulas expressing axioms and theorems. CHRs can also be found by first considering special cases of each constraint and then looking at interactions of pairs of constraints sharing a variable. Cases that don't occur in the application can be ignored. CHRs can also improve application programs by turning certain predicates into constraints to provide "short-cuts" (lemmas). For example, to the predicate `append/3` one can add `append(L1,[],L2) <=> L1=L2` together with `label_with append(L1,L2,L3) if true`.
Starting from an executable specification, the rules can then be refined and adapted to the specifics of the application. *Efficiency can be improved* by strengthening or weakening the

guards to perform simplification as early as needed and to do the "just right" amount of propagation. Propagation rules can be expensive, because no constraints are removed. If the speed of the final handler is not satisfactory, it can be rewritten using meta-terms or auxiliary C functions.
The rules for a constraint can be scattered across the `chr` file as long as they are in the same module. The rules are tried in *some order* determined by the CHR compiler. Due to optimizations this order is not necessarily the textual order in which the rules where written. In addition, the incremental addition of constraints at run-time causes constraints to be tried for application of rules in some dynamically determined order.

6.7.2 Optimizations

Single-headed rules should be preferred to two-headed rules which involve the expensive search for a partner constraint. Rules with *two heads can be avoided* by changing the "granularity" of the constraints. For example, assume one wants to express that n variables are different from each other. It is more efficient to have a single constraint `all_different(List_of_n_Vars)` than n^2 inequality constraints (see handler `domain.chr`). However, the extreme case of having a single constraint modeling the whole constraint store will usually be inefficient.
Rules with two heads are more efficient, if the two heads of the rule share a variable (which is usually the case). Then the search for a partner constraint has to consider less candidates. Moreover, two rules with identical (or sufficiently similar) heads can be merged into one rule so that the search for a partner constraint is only performed once instead of twice.
Rules with more than two heads are not allowed for efficiency reasons. If needed, they can usually be written as several rules with two heads. For example, in the handler for set constraints `set.chr`, the propagation rule:

```
set_union(S1, S2, S), set(S1, S1Glb, S1Lub), set(S2, S2Glb, S2Lub) ==>
        s_union(S1Glb, S2Glb, SGlb),
        s_union(S1Lub, S2Lub, SLub),
        set(S, SGlb, SLub).
```

is translated into:

```
set_union(S1, S2, S), set(S1, S1Glb, S1Lub) ==>
        '$set_union'(S2, S1, S1Glb, S1Lub, S).
set(S2, S2Glb, S2Lub) \ '$set_union'(S2, S1, S1Glb, S1Lub, S) <=>
        s_union(S1Glb, S2Glb, SGlb),
        s_union(S1Lub, S2Lub, SLub),
        set(S, SGlb, SLub).
```

As *guards* are tried frequently, they should be simple *tests* not involving side-effects. For efficiency and clarity reasons, one should also avoid using user-defined constraints in guards. Currently, besides conjunctions, disjunctions are allowed in the guard, but they should be used with care. The use of other control built-in predicates of ECL^{i}PS^e is discouraged. Negation and if-then-else can be used if their first arguments are either *simple goals* (see ECL^{i}PS^e user manual) or goals that don't touch global variables. Similarly, goals preceding a cut must fulfill this condition. *Built-in constraints* (e.g. finite domains, rational arithmetic) work as tests only in the current implementation. Head matching is more efficient than explicitly checking equalities in the guard (which requires the `check_guard_bindings` flag to be on).

In the current implementation, local variables (those that do not occur in the heads) can be shared between the guard and the body.
Several handlers can be used simultaneously if they don't share user-defined constraints. The current implementation will not work correctly if the same constraint is defined in rules of different handlers that have been compiled separately. In such a case, the handlers must be merged "by hand". This means that the source code has to be edited so that the rules for the shared constraint are together (in one module). Changes may be necessary (like strengthening guards) to avoid divergence or loops in the computation.
Constraint handlers can be tightly integrated with constraints defined with *other extensions of ECL^{i}PS^e* (e.g. meta-terms) by using the ECL^{i}PS^e built-in predicate `notify_constrained(Var)` to notify ECL^{i}PS^e each time a variable becomes more constrained. This happens if a user-defined constraint is called for the first time or if a user-defined constraint is rewritten by a CHR into a stronger constraint with the same functor.
For *pretty printing* of the user-defined constraints in the answer at the top-level and debuggers, ECL^{i}PS^e macro transformation (for write mode) can be used. This is especially useful when the constraints have some not so readable notation inside the handler. For an example, see the constraint handler bool `bool.chr`.

6.8 Debugging CHR Programs

User-defined constraints including application of CHRs can be traced with the standard debugger. Debugging of the ECL^{i}PS^e code is done in the standard way. See the corresponding user manual for more information.

6.8.1 Using the Debugger

In order to use the debugging tool, the `debug_compile` flag must have been `on` (default) during compilation (`chr` to `pl`) and loading of the produced ECL^{i}PS^e code.

- The query `trace.` activates the standard debugger (tracing user-defined constraints like predicates).
- The query `chr_trace.` activates the standard debugger showing more information about the handling of constraints. (application of CHRs).
- The query `chr_notrace.` stops either debugger.

The debugger displays user-defined constraints and application of CHRs. User-defined constraints are treated as predicates and the information about application of CHRs is displayed without stopping. See the subsection on how CHRs work for an example trace. The additional ports are:

- `add`: A new constraint is added to the constraint store.
- `already_in`: A constraint to be added was already present.

The ports related to application of rules are:

- `try_rule`: A rule is tried.

- `delay_rule`: The last tried rule cannot fire because the guard did not succeed.
- `fire_rule`: The last tried rule fires.

The ports related to labeling are:

- `try_label`: A label_with declaration is checked.
- `delay_label`: The last label_with declaration delays because the guard did not succeed.
- `fire_label`: The last tried label_with declaration succeeds, so the clauses of the associated constraint will be used for built-in labeling.

When displayed, each constraint is labeled with a unique integer identifier. Each rule is labeled with its name as given in the `chr` source using the `@` operator. If a rule does not have a name, it is displayed together with a unique integer identifier.

6.9 The Extended CHR Implementation

A new, extended, `chr` library has been developed, with the intention of providing the basis for a system that will allow more optimisations than the previous implementation. At the same time, some of the syntax of the CHR has been changed to conform better to standard Prolog.
The system is still experimental, and provides no special support for debugging CHR code. Please report any problems encountered while using this system.
The main user visible differences from the original `chr` library are as follows:

- The extended library produces code that generally runs about twice as fast as the old non-debugging code. It is expected that further improvements should be possible.
- CHR code is no longer compiled with a special command – the normal compile command will now recognise and compile CHR code when the extended `chr` library is loaded. No intermediate Prolog file is produced. The `.chr` extension is no longer supported implicitly.
- Syntax of some operators have been changed to conform better to standard Prolog.
- A framework for supporting more than two head constraints has been introduced. However, support for propagation rules with more than two heads have not yet been added. Simplification and simpagation rules with more than two heads are currently supported.
- The compiler does not try to reorder the CHR any more. Instead, they are ordered in the way they are written by the user.
- `label_with` is no longer supported. It can be replaced with user defined labelling.
- The operational semantics of rules have been clarified.
- There is no special support for debugging yet. The CHR code would be seen by the debugger as the transformed Prolog code that is generated by the compiler.

6.9.1 Invoking the extended CHR library

The extended library is invoked by `lib(ech)`. Given that it is now integrated into the compiler. It can be invoked from a file that contains CHR code, as `:- lib(ech).`, as long as this occurs before the CHR code.

6.9.2 Syntactic Differences

As programs containing CHRs are no longer compiled by a separate process, the `.chr` extension is no longer implicitly supported. Files with the `.chr` extension can still be compiled by explicitly specifying the extension in the compile command, as in `['file.chr']`. Associated with this change, there are some changes to the declarations of the `.chr` format:

- `operator/3` does not exist. It is not needed because the standard Prolog `op/3` declaration can now handle all operator declarations. Replace all `operator/3` with `op/3` declarations.
- The other declarations `handler constraints option` are now handled as normal Prolog declarations, i.e. they must be preceded with `:-`. This is to conform with standard Prolog syntax.

The syntax for naming a rule has been changed, because the old method (using `@` clashes with the use of `@` in modules. The new operator for naming rules is `::=`. Here is part of the minmax handler in the new syntax:

```
:- handler minmax.
:- constraints leq/2, neq/2, minimum/3, maximum/3.
:- op(700, xfx, leq).

built_in    ::=  X leq Y <=> \+nonground(X), \+nonground(Y) | X @=< Y.
reflexivity ::=  X leq X <=> true.
...
```

6.9.3 Compiling

After loading the extended `chr` library, programs containing CHR code can be compiled directly. Thus, CHR code can be freely mixed with normal Prolog code in any file. In particular, a compilation may now compile code from different files in different modules which may all contain CHR codes. This was not a problem with the old library because CHR code had to be compile separately.
In the extended library, CHR code can occur anywhere in a particular module, and for each module, all the CHR code (which may reside in different files) will all be compiled into one unit (`handler` declarations are ignored by the system, they are present for compatibility purposes only), with the same constraint store. CHR code in different modules are entirely separate and independent from each other.
In order to allow CHR code to occur anywhere inside a module, and also because it is difficult to define a meaning for replacing multi-heads rules, compilation of CHR code is always incremental, i.e. any existing CHR code in a module is not replaced by a new compilation. Instead, the rules from the new compilation is added to the old ones.

It is possible to clear out old CHR code before compiling a file. This is done with the `chr/1` predicate. This first remove any existing CHR code in any module before the compilation starts. It thus approximates the semantics of `chr/1` of the old library, but no Prolog file is generated.

6.9.4 Semantics

6.9.4.1 Addition and removal of constraints

In the old `chr` library, it was not clearly defined when a constraint will be added to or removed from the constraint store during the execution of a rule. In the extended `chr` library, all head constraints that occur in the head of a rule are mutually exclusive, i.e. they cannot refer to the same constraint. This ensures that similar heads in a rule will match different constraints in the constraint store. Beyond this, the state of a constraint – if it is in the constraint store or not – that has been matched in the head is not defined during the execution of the rest of the head and guard. As soon as the guard is satisfied, any constraints removed by a rule will no longer be in the constraint store, and any constraint that is not removed by the rule will be present in the constraint store.
This can have an effect on execution. For example, in the finite domain example in the old `chr` directory (`domain.chr`), there is the following rule:

```
X lt Y, X::[A|L] <=>
        \+nonground(Y), remove_higher(Y,[A|L],L1), remove(Y,L1,L2) |
        X::L2.
```

Unfortunately this rule is not sufficiently specified in the extended CHR, and can lead to looping under certain circumstances. The two `remove` predicate in the guard removes elements from the domain, but if no elements are removed (because `X lt Y` is redundant, e.g. `X lt 5` with `X::[1..2]`), then in the old CHR execution, the body goal, the constraint `X::L2` would not be actually executed, because the older constraint in the head (the one that matched `X::[A|L]`) has not yet been removed when the new constraint is imposed. With the extended CHR, the old constraint is removed after the guard, so the `X::L2` is executed, and this can lead to looping. The rule should thus be written as:

```
X lt Y, X::[A|L] <=>
        \+nonground(Y), remove_higher(Y,[A|L],L1), remove(Y,L1,L2),
                L2\==[A|L] |
        X::L2.
```

6.9.4.2 Executing Propagation and simpagation rules

Consider the following propagation rule:

```
p(X), q(Y) ==> <Body>.

:- p(X).
```

The execution of this rule, started by calling `p(X)`, will try to match all `q(Y)` in the constraint store, and thus it can be satisfied, with `<Body>` executed, multiple number of times with different `q(Y)`. `<Body>` for a particular `q(Y)` will be executed first, before trying to match the next `q(Y)`. The execution of `<Body>` may however cause the removal of `p(X)`. In this case, no further matching with `q(Y)` will be performed.
Note that there is no commitment with propagation and simpagation rule if the constraint being matched is not removed:

```
p(X), q(Y) ==> <Body1>.
p(X), r(Y) ==> <Body2>.

:- p(X).
```

Both rules will always be executed.
The body of a rule is executed as soon as its guard succeeds. In the case of propagation rules, this means that the other propagation rules for this constraint will not be tried until the body goals have all been executed. This is unlike the old CHR, where for propagation rules, the body is not executed until all the propagation rules have been tried, and if more than one propagation rule has fired (successful in its guard execution), then the most recently fired rule's body is executed first. For properly written, mutually exclusive propagation rule, this should not make a difference (modulo the effect of the removal of constraints in the body).

6.9.5 Options and Built-In Predicates

The `check_guard_bindings` and `already_in_store` options from the old `chr` library are supported. Note that the extended compiler can actually detect some cases where guard bindings cannot constrain any global variables (for example, `var/1`), and will in such cases no check guard bindings.
New options, intended to control the way the compiler tries to optimise code, are introduced. These are intended for the developers of the compiler, and will not be discussed in detail here. The only currently supported option in this category is `single_symmetric_simpagation`.
The old CHR built-ins, `chr_get_constraint/1` and `chr_get_constraint/2` are both implemented in this library.
A new built-in predicate, `in_chrstore/1`, is used to inspect the constraint store:

```
in_chrstore(+Constraint)
```

is used to test if `Constraint` is in the constraint store or not. It can be used to prevent the addition of redundant constraints:

```
X leq Y, Y leq Z ==> \+in_chrstore(X leq Z)| X leq Z.
```

The above usage is only useful if the `already_in_store` option is off. Note that as the state of a constraint that appears in the head is not defined in the guard, it is strongly suggested that the user does not perform this test in the guard for such constraints,

6.9.6 Compiler generated predicates

A source to source transformation is performed on CHR code by the compiler, and the resulting code is compiled in the same module as the CHR code. These transformed predicates all begin with 'CHR', so the user should avoid using such predicates.

Chapter 7

RANGE: A Basis For Numeric Solvers

7.1 Introduction

This library implements variables that range over integer or real intervals. It is meant to be used as a common basis for arithmetic constraint solvers, and it can serve as a mechanism to make such solvers communicate.

7.2 Usage

Load the library by using

```
:- lib(range).
```

You will need ECL^{i}PS^e version 3.5.1 or higher.

7.3 Library Predicates

7.3.1 Constraints

Important hint: All constraints in this libary may trigger waking when applied to existing variables. In this case, **schedule_suspensions/1** will be executed by the predicate, so the goals will be scheduled for waking, but not actually executed. The caller therefore has to call **wake/0** (the woken goal scheduler) at an appropriate point in the subsequent execution.

7.3.1.1 Vars :: Lo..Hi

Logically: Constrain a variable (or all variables in a list) to take only integer or real values in a given range. The type of the bounds determines the type of the variable (real or integer). Also allowed are the (untyped) symbolic bound values `inf`, `+inf` and `-inf`. For instance

```
X :: 0..1          % boolean
X :: -1..5         % integer between -1 and 5
X :: 1..inf        % strictly positive integer
X :: 0.0..10.0     % real between 0.0 and 10.0
```

```
X :: 1.5..3.7       % real between 1.5 and 3.7
X :: 0.0..inf       % positive real
X :: 0.0..5         % TYPE ERROR
```

Operationally, the range and type information is immediately stored into the variable's attribute.

7.3.1.2 reals(Vars)

The domain of the variables is the real numbers. This is the default, so the declaration is optional. `real(X)` is equivalent to `X :: -inf..inf`. Mathematical Programming style nonnegative variables are best declared as `X :: 0.0..inf`.
Note that the notion of real numbers is used here in the pure mathematical sense, where real numbers subsume the integers. A variable of type `real` can therefore be instantated to either a floating point or an integer number.

7.3.1.3 integers(Vars)

Constrain the variables to integer values. Note that this declaration is implicit when specifiying an integer range, e.g. in `Y :: 0..99`.

7.3.1.4 lwb(+Var, +Bound)

Constrain the variable to be greater or equal to the specified lower bound. A bound update on a variable may fail (when the update empties the domain), succeed (possibly updating the variable's bounds), or instantiate the variable (in case the domain get restricted to a singleton value). Note that if the variable's type is integer, its bounds will always be adjusted to integral values.

7.3.1.5 upb(+Var, +Bound)

Constrain the variable to be less or equal to the specified upper bound.

7.3.1.6 Examples

Every new constraint on a variable is immediately reflected in the range:

```
[eclipse 2]: X::0.0..9.5, lwb(X,4.5).
X = X{4.5 .. 9.5}
yes.
[eclipse 3]: X::0.0..9.5, lwb(X,4.5), integers([X]).
X = X{5 .. 9}
yes.
[eclipse 4]: X::0.0..9.5, lwb(X,4.5), integers([X]), upb(X,5.9).
X = 5
yes.
[eclipse 5]: X::0.0..9.5, lwb(X,4.5), upb(X,4.3).
no (more) solution.
```

7.3.2 Retrieving Domain Information

7.3.2.1 var_range(+Var, -Lo, -Hi)

Retrieve the current range of a variable (or number). Lo and Hi return the minimum and maximum (respectively) of the variable's range in floating point format (regardless of the variable's type). If Var has not been declared before, it will be turned into an unrestricted real variable. If Var is a number, that number will be returned as both Lo and Hi.

7.3.2.2 var_type(+Var, -Type)

Retrieve the type ('real' or 'integer') of a variable (or number).

7.3.3 Auxliliary Predicates

7.3.3.1 range_msg(+Var1, +Var2, ?Var3)

The most specific generalisation of two ranges is computed and returned as Var3. Var3 will range over the smallest interval enclosing the input ranges, and have the more general type of the inpout types.

7.3.3.2 print_range(+Var, -Range)

Returns the variable's range in a form that would be acceptable to ::/2, ie. as a Lo..Hi structure, encoding the variable's type in the type of the bounds.

7.3.4 Handlers

The library installs the following handlers (cf. ECL^{i}PS^{e}User Manual) in order to implement the semantics of ranged variables:

unify Unification between two variables amounts to intersecting their ranges and taking the more restrictive type as the result type. If the intersection is empty, the unification fails. Unifying a variable with a number involves a check whether the number is within the variable's range and of the proper type, otherwise failure occurs.

test_unify like unify.

compare_instances A range variable is an instance of another when its range is subsumed by the other range.

copy_term Range and type are copied, delayed goals are not.

delayed_goals, delayed_goals_number Considers the goals in the two attached suspension lists.

print Ranges are printed using print_range/2.

Due to the handlers, Unification and instance test take the ranges into account:

```
[eclipse 6]: X::0.0..5.5, Y::3..8, X=Y.
X = X{3 .. 5}
Y = X{3 .. 5}
yes.
[eclipse 8]: X::0.0..5.5, Y::3..8, instance(X,Y).
no (more) solution.
[eclipse 9]: X::0.0..5.5, Y::3..5, instance(Y,X).
Y = Y{3 .. 5}
X = X{0.0 .. 5.5}
yes.
```

7.4 Attribute Structure

Ranged variables are implemented as attributed variables. The attribute contains the following fields:

type specifies the variable type, either integer or real.

lo the smallest value the variable can assume

hi the largest value the variable can assume

wake_lo list of goals to be woken on lower bound change

wake_hi list of goals to be woken on upper bound change

Type and bounds are accessed through the predicates described above. Goals can be delayed on the waking lists using the suspend/3 predicate, for example:

```
[eclipse 13]: X::0.0..5.5, suspend(writeln(change), 3, X->wake_lo), lwb(X,1).
change
X = X{1.0 .. 5.5}
yes.
```

7.5 Writing Higher Level Constraints

The following example can be taken as a scheme for how to write constraints on top of the facilities of this library. It is a greater-equal constraint for two variables:

```
ge(X, Y) :-                     % woken on change of bounds
    var_range(X, _, XH),
    var_range(Y, YL, _),
    ( var(X),var(Y) ->
        suspend(ge(X,Y), 3, [X->wake_hi, Y->wake_lo])
    ;
        true),
    lwb(X, YL), upb(Y, XH),     % impose new bounds
    wake.                       % execute woken goals here
```

The constraint wakes when either the upper bound of X or the lower bound of Y changes, and imposes the consequences onto the other variable. When lwb/2 and upb/2 cause further bound changes, that may wake other goals (ie. they have the effect of **schedule_suspensions/1**) and we therefore have to invoke the waking scheduler **wake/0** afterwards.

Chapter 8

EPLEX: The ECL^{i}PS^e/CPLEX Interface

8.1 Usage

This library lets you use an external Simplex or MIP solver like CPLEX[1] or XPRESS-MP[2] from within ECL^{i}PS^e. Load the library by using either of

```
:- lib(eplex_cplex).
:- lib(eplex_xpress).
:- lib(eplex).
```

The first line explicitly requests the CPLEX solver, the second line explicitly requests the XPRESS-MP solver, and the third line will try to load whatever licenced solver is available on the computer. Note that the eplex-library provides a largely solver-independent API to the programmer, so many programs will run with either external solver.

8.2 Versions and Licences

Note that the ECL^{i}PS^e library described here is just an interface to CPLEX or XPRESS-MP. In order to be able to use it, you need to have a licence for one of these solvers on your machine.
Depending on whether you have CPLEX or XPRESS-MP, which version of it, and which hardware and operating system, you need to use the matching version of this interface. Because an ECL^{i}PS^e installation can be shared between several computers on a network, we have provided you with the possibility to tell the system which licence you have on which machine. To configure your local installation, simply add one line for each computer with a CPLEX or XPRESS-MP licence to the file `<eclipsedir>/lib/eplex_lic_info.ecl`, where `<eclipsedir>` is the directory or folder where your ECL^{i}PS^e installation resides. For example, if you have CPLEX version 6.5 on machine `workhorse`, you would add the line

```
licence(workhorse, cplex, '65', '', 0).
```

Note that the set of supported solver versions may vary between different releases of ECL^{i}PS^e.

[1] CPLEX is a registered trademark of CPLEX Optimization Inc.
[2] XPRESS-MP is a product from Dash Associates Ltd.

8.3 Ranged and Typed Variables

Ranged variables are provided by the range-library. The relevant predicate are:

Vars :: Lo..Hi Define the inital bounds of variables. Note that if both bounds are specified as integers, the variable will be an integer one. E.g. X::1..9 declares an integer variable while X::1.0..9.0 declares a continuous one. The symbolic bounds -inf and inf can be used. The default range is -inf..inf. Mathematical Programming style nonnegative variables should be declared as `X :: 0.0..inf`.

reals(Vars) Equivalent to `X :: -inf..inf`.

integers(Vars) Constrain the variables to integer values.

var_range(+Var, -Lo, -Hi) Retrieve a variable's range.

var_type(+Var, -Type) Retrieve a variable's type (integer or real).

8.4 Black-Box Interface

One possible use of this library is to use ECL^{i}PS^{e}just as a modeling language and let the external solver do all the solving. For that purpose, a high-level interface is provided. It consists of primitives for setting up linear constraints and a single optimization primitive to invoke the external solver on these constraints.

8.4.1 Linear Constraints

The constraints provided are equalities and inequalities over linear expressions. Their operational behaviour is as follows:

- When they contain no variables, they simply succeed or fail.
- When they contain exactly one variable, they are translated into a bound update on that variable, which may in turn fail, succeed, or even instantiate the variable. Note that if the variable's type is integer, the bound will be adjusted to the next suitable integral value.
- Otherwise, the constraint delays until it is later transferred to the external solver. This mechanism makes it possible to interface to a non-incremental black-box solver that requires all constraints at once, or to send constraints to the solver in batches

8.4.1.1 X $= Y

X is equal to Y. X and Y are linear expressions.

8.4.1.2 X $>= Y

X is greater or equal to Y. X and Y are linear expressions.

8.4.1.3 X $=< Y

X is less or equal to Y. X and Y are linear expressions.

8.4.2 Linear Expressions

The following arithmetic expression can be used inside the constraints:

X Variables. If X is not yet a ranged variable, it is turned into one via an implicit declaration `X :: -inf..inf`.

123, 3.4 Integer or floating point constants.

+Expr Identity.

-Expr Sign change.

E1+E2 Addition.

sum(ListOfExpr) Equivalent to the sum of all list elements.

E1-E2 Subtraction.

E1*E2 Multiplication.

ListOfExpr1*ListOfExpr2 Scalar product: The sum of the products of the corresponding elements in the two lists. The lists must be of equal length.

8.4.3 Optimization

After setting up the constraints with the primitives described above, the external solver's MIP optimizer can be invoked as a black box using

8.4.3.1 optimize(+Objective, -Cost)

Objective is either min(Expr) or max(Expr) where Expr is a linear expression. This calls the external solver's optimizer and succeeds if it finds an optimum. In this case the problem variables get instantiated to the solution values, and Cost gets bound to the cost of this solution. Note that this will find at most one solution, ie. you won't get alternative optima on backtracking.
In section 8.6.6 we will later show how optimize/2 is built on top of the lower level functionality.

8.4.4 Examples

Here is a simple linear program. As long as the optimizer is not invoked, the constraints just delay:

```
[eclipse 2]: X+Y $>= 3, X-Y $= 0.
X = X
Y = Y
Delayed goals:
```

```
        X + Y$>=3
        X - Y$=0
yes.
```

Now a call to optimize/2 is added in order to trigger the solver:

```
[eclipse 3]: X+Y $>= 3, X-Y $= 0, optimize(min(X), C).

Y = 1.5
X = 1.5
C = 1.5
```

(Note that X and Y have not been explicitly declared. They default to reals ranging from -infinity to +infinity.)
By declaring one variable as integer, we obtain a Mixed Integer Problem:

```
[eclipse 4]: integers([X]), X+Y $>= 3, X-Y $= 0, optimize(min(X), C).

Y = 2.0
X = 2
C = 2.0
yes.
```

8.5 Interface for CLP-Integration

8.5.1 Simplex Demons

To implement hybrid algorithms where a run of a simplex solver is only a part of the global solving process, the black-box model presented above is not appropriate any more. As a more convenient model, we introduce the concept of a simplex demon. A simplex demon collects linear constraints and re-solves the problem whenever bounds change or new constraints appear.

8.5.1.1 lp_demon_setup(+Objective, -Cost, +ListOfOptions, +Priority, +TriggerModes, -Handle)

Declaratively, this can be seen as a compound constraint representing all the individual linear constraints that have been set so far and are going to be set up later. Operationally, the delayed constraints are collected and an external solver is set up (as with lp_setup/4). Then the problem is solved once initially (as with lp_solve/2) and a delayed goal lp_demon/7 is set up which will re-trigger the solver when certain conditions are met.
`Handle` refers to the created solver state (as in lp_setup/4 or lp_read/3 described below). It can be used to access and modify the state of the solver, retrieve solution information etc.
Unlike with lp_solve/2, `Cost` will not be instantiated to a solution's cost, but only be bounded by it: For a minimization problem, each solution's cost becomes a lower bound, for maximization an upper bound on Cost. This technique allows for repeated re-solving with reduced bounds or added constraints.
`ListOfOptions` is a list of solver options as described is section 8.6.1.1 for lp_setup/4.

`Priority` is the scheduling priority with which the solver gets woken up. This priority determines whether the solver is run before or after other constraints. It is recommended to choose a priority that lies below the priority of more efficient propagation constraints, e.g. 5. `TriggerModes` specifies under which conditions the solver demon will be re-triggered. It can be a list of the following specifiers

`inst`: re-trigger if a problem variable gets instantiated.

`deviating_inst`: re-trigger if a problem variable gets instantiated to a value that differs from its lp-solution more than a tolerance.

`bounds`: re-trigger each time a variable bound changes.

`deviating_bounds`: re-trigger each time a variable bound changes such that its lp-solution gets excluded more than a tolerance.

`new_constraint`: re-trigger each time a new constraint appears.

`trigger(Atom)`: re-trigger each time the symbolic trigger Atom is pulled by invoking **schedule_suspensions/1**.

`pre(Goal)`: an additional condition to be used together with other triggers. When the demon is triggered, it first executes `PreGoal`. Only if that succeeds, does the appropriate external solver get invoked. This provides a way of reducing the number of (possibly expensive) solver invocations when given preconditions are not met.

`post(Goal)`: this is not a trigger condition, but specifies a goal to be executed after solver success, but before the Cost variable gets constrained. It is intended as a hook for exporting solution information, e.g. copying solutions from the solver state into variable attributes (eg. tentative value), or computing weights for labeling heuristics from the solver state.

The tolerances mentioned can be specified in lp_setup/2 or lp_set/3 as `demon_tolerance`.
Some common invocations patterns for this predicate are the following. The first triggers the solver only on instantiation of variables to values that don't fit with the simplex solution:

```
lp_demon_setup(min(Expr), C, [], 5, [deviating_inst], H)
```

The next one is more eager and triggers on significant bound changes or whenever new constraints arrive:

```
lp_demon_setup(max(Expr), C, [], 5, [new_constraint,deviating_bounds], H)
```

The solver can also be triggered explicitly by setting it up with

```
lp_demon_setup(min(Expr), C, [], 5, [trigger(run_simplex)], H)
```

and then issuing the command

```
schedule_suspensions(run_simplex),wake
```

If several trigger conditions are specified, then any of them will trigger the solver.
When a solver demon runs frequently on relatively small problems, it can be important for efficieny to switch the external solver's presolving off (lp_set(presolve,0)) to reduce overheads.

8.5.1.2 solution_out_of_range(+Handle)

This is intended as a useful pre(Goal) for lp_demon_setup/6 in connection with the `bounds` trigger mode. It succeeds if any of the solutions (computed by the most recent successful solving) of Handle are more than a tolerance outside the range of the corresponding variables, ie. couldn't be instantiated to this value. The admissible tolerances can be specified in lp_setup/2 or lp_set/3 as `demon_tolerance`.

8.5.1.3 instantiation_deviates(+Handle)

This is intended as a useful pre(Goal) for lp_demon_setup/6 in connection with the `inst` trigger mode. It succeeds if any of the variables originally involved in Handle have been instantiated to a value that is not within +/− tolerance from the latest simplex solution for that variable. The admissible tolerances can be specified in lp_setup/2 or lp_set/3 as `demon_tolerance`.

8.5.2 Example

The simplest case of having a simplex solver automatically cooperating with a CLP program, is to set up a solver demon which will repeatedly check whether the continuous relaxation of a set of constraints is still feasible. The code could look as follows:

```
simplex :-
    lp_demon_setup(min(0), C, [solution(no)], 5, [bounds], _).
```

First, the constraints are normalised and checked for linearity. Then a solver with a dummy objective function is set up. The option `solution(no)` indicates that we are not interested in solution values. Then we start a solver demon which will re-examine the problem whenever a change of variable bounds occurs. The demon can be regarded as a compound constraint implementing the conjunction of the individual constraints. It is able to detect some infeasibilities that for instance could not be detected by the finite domains solver, e.g.

```
[eclipse]: X+Y+Z $>= K, X+Y+Z $=< 1,
    lp_demon_setup(min(0), C, [solution(no)], 5, [bounds], _),
    K = 2.

no (more) solution.
```

In the example, the initial simplex is successful, but instantiating K wakes the demon again, and the simplex fails this time.
A further step is to take advantage of the cost bound that the simplex procedure provides. The setup is similar to above, but we accept an objective function and add a cost variable. The bounds of the cost variable will be updated whenever a simplex invocation finds a better cost bound on the problem. In the example below, an upper bound for the cost of 1.5 is found initially:

```
[eclipse 14]: X+Y $=< 1, Y+Z $=< 1, X+Z $=< 1,
        lp_demon_setup(max(X+Y+Z), Cost, [solution(no)], 5, [bounds], _).

X = X{-1e+20 .. 1e+20}
```

```
Y = Y{-1e+20 .. 1e+20}
Z = Z{-1e+20 .. 1e+20}
Cost = Cost{-1e+20 .. 1.500001}

Delayed goals:
        lp_demon(prob(...), ...)
yes.
```

If the variable bounds change subsequently, the solver will be re-triggered, possibly improving the cost bound to 1.3:

```
[eclipse 16]: X+Y $=< 1, Y+Z $=< 1, X+Z $=< 1,
        lp_demon_setup(max(X+Y+Z), Cost, [solution(no)], 5, [bounds], _),
        Y $=< 0.3.

X = X{-1e+20 .. 1e+20}
Z = Z{-1e+20 .. 1e+20}
Cost = Cost{-1e+20 .. 1.300001}
Y = Y{-1e+20 .. 0.3}

Delayed goals:
        lp_demon(prob(...), ...)
yes.
```

A further example is the implementation of a MIP-style branch-and-bound procedure. Source code is provided in the library file mip.pl.

8.6 Low-Level Solver Interface

For many applications, the facilities presented so far should be appropriate for using Simplex/MIP through ECL^{i}PS^e. This section describes lower level operations like how to set up solvers manually and the primitives available to access and modify a solver's state.

8.6.1 Setting up Solvers Manually

This basic interface allows the user to deal with several independent solvers, to set them up, solve and re-solve, extract information about a solver's state and modify various parameters. Each such solver is referred to by a handle representing the solver's state.

8.6.1.1 lp_setup(+NormConstraints, +Objective, +ListOfOptions, -Handle)

Create a new solver state for the set of constraints NormConstraints (see below for how to obtain a set of normalised constraints). Apart from the explicitly listed constraints, the variable's ranges will be taken into account as the variable bounds for the simplex algorithm. Undeclared variables are implicitly declared as reals/1.
However, when variables have been declared integers (using ::/2 or integers/1), that is not taken into account by the solver by default. This means that the solver will only work on the *relaxed problem* (ie. ignoring the integrality constraints), unless specified otherwise in

the options. Objective is either `min(Expr)` or `max(Expr)` where Expr is a linear expression. Options is a list of options (see below). A solver-handle is returned which is used to refer to the solver subsequently.
The solver Options are:

`integers(all)` Advises the solver to take all integrality constraints into account, ie. to consider all variables integers that have been declared such. This option will instruct the external solver to use its own MIP solver (ie. branch-and-bound search happens within the external solver) instead of just the Simplex.

`integers(+ListOfVars)` Consider the specified variables to be integers (whether or not they have been declared such). This option will instruct the external solver to use its own MIP solver (ie. branch-and-bound search happens within the external solver) instead of just the Simplex.

`method(+Method)` Use the specified method (`primal, dual, netprimal, netdual, barrier`) to solve the problem. The default is `primal`. See th external solver's manual for a description of these methods.

`solution(+YesNo)` Make the solutions available each time the problem has been (re-)solved successfully. YesNo is one of the atoms `yes` or `no`, the default is `yes`.

`dual_solution(+YesNo)` Make the dual solutions available each time the problem has been (re-)solved successfully. YesNo is one of the atoms `yes` or `no`, the default is `no`.

`slack(+YesNo)` Make the constraint slacks available each time the problem has been (re-)solved successfully. YesNo is one of the atoms `yes` or `no`, the default is `no`.

`reduced_cost(+YesNo)` Make the reduced costs available each time the problem has been (re-)solved successfully. YesNo is one of the atoms `yes` or `no`, the default is `no`.

`keep_basis(+YesNo)` Store the basis each time the problem has been solved successfully, and use this basis as a starting point for re-solving next time. This option only affects performance. YesNo is one of the atoms `yes` or `no`, the default is `no`.

`demon_tolerance(RealTol, IntTol)` Specify how far outside a variable's range an lp-solution can fall before lp_demon_setup/6 re-triggers. `RealTol` and `IntTol` are floats and default to 0.00001 and 0.5 respectively.

`simplify(+YesNo)` Simplify the constraints before sending them to the external solver. The simplification consists of eliminating trivial constraints and turning simple constraints into bound updates. It is solver-dependent whether this step is needed or whether it is covered by the external solvers's preprocessing. YesNo is one of the atoms `yes` or `no`, the default is `no`.

`space(+Rows,+Cols,+NonZeros)` This option is needed with solvers that require a-priory memory allocation (currently only XPRESS-MP). The arguments are integers specifying how many extra rows, columns and nonzero coefficients can be added to the solver after it has been set up.

8.6.1.2 lp_set(+Handle, +What, +Value)

This primitive can be used to change some of the initial options even after setup. *Handle* refers to an existing solver state, *What* can be one of the following:

`method` Set the method that will be used to solve the problem. Value is one of `primal, dual, netprimal, netdual, barrier`.

`solution` Make the solutions available each time the problem has been (re-)solved successfully. Value is one of the atoms `yes` or `no`.

`reduced_cost` Make the reduced costs available each time the problem has been (re-)solved successfully. Value is one of the atoms `yes` or `no`.

`slack` Make the constraint slacks available each time the problem has been (re-)solved successfully. Value is one of the atoms `yes` or `no`.

`dual_solution` Make the dual solutions available each time the problem has been (re-)solved successfully. Value is one of the atoms `yes` or `no`.

`keep_basis` Store the basis each time the problem has been solved successfully, and use this basis as a starting point for re-solving next time. Value is one of the atoms `yes` or `no`.

`demon_tolerance` Specify how far outside a variable's range an lp-solution can fall before lp_demon_setup/6 re-triggers. Value is a comma-separated pair `(RealTol,IntTol)` of floating-point values (default `(0.00001,0.5)`).

`simplify` Simplify the constraints before sending them to the external solver. The simplification consists of eliminating trivial constraints and turning simple constraints into bound updates. It is solver-dependent whether this step is needed or whether it is covered by the external solvers's preprocessing. Value is one of the atoms `yes` or `no`.

Making solutions available means that they can be retrieved using lp_get/3 or lp_var_get/3 after the solver has been run successfully.

8.6.1.3 lp_add(+Handle, +NewNormConstraints, +NewIntegers)

Add new constraints (with possibly new variables) to a solver. The new constraints will be taken into account the next time the solver is run. The constraints will be removed on backtracking.

8.6.1.4 lp_cleanup(+Handle)

Destroy the specified solver, free all memory, etc. Note that ECL^{i}PS^{e}will normally do the cleanup automatically, for instance when execution fails across the solver setup, or when a solver handle gets garbage collected. However, calling lp_cleanup/1 explicitly does not hurt and may cause resources (memory and licence) to be freed earlier.

8.6.1.5 lp_read(+File, +Format, -Handle)

Read a problem from a file and setup a solver for it. Format is `lp` or `mps`. The result is a handle similar to the one obtained by lp_setup/4.

8.6.1.6 lp_write(+Handle, +Format, +File)

Write the specified solver's problem to a file. Format is `lp` or `mps`.

8.6.2 Running a Solver Explicitly

A solver needs to be triggered to actually solve the Linear Programming or Mixed Integer Programming problem that it represents. While solvers created by optimize/2 and lp_demon_setup/6 are triggered automatically, solvers that have been set up manually with lp_solve/2 need to be run explicitly.

8.6.2.1 lp_solve(+Handle, -Cost)

Apply the external solver's LP or MIP solver to the problem represented by Handle. Precisely which method is used depends on the options given to lp_setup/4. lp_solve/2 fails if there is no solution or succeeds if an optimal solution is found, returning the solution's cost in Cost (unlike with lp_demon_setup/6, Cost gets instantiated to a number). After a success, various solution and status information can be retrieved using lp_get/3,4.
If there was an error condition, or limits were exceeded, lp_solve/2 raises the error 'CPLEX_ABORT'. Even in that case, the external solver return status can be obtained using `lp_get(Handle, status, ...)`.
When a solver is triggered repeatedly, each invocation will automatically take into account the current variable bounds. The set of constraints considered by the solver is the one given when the solver was created plus any new constraints that were added (lp_add/3) in the meantime.

8.6.2.2 lp_probe(+Handle, +Objective, -Cost)

Similar to lp_solve/2, but optimize for a different objective function rather than the one that was specified during solver setup.

8.6.3 Accessing Solutions and other Solver State

8.6.3.1 lp_get(+Handle, +What, -Value)

Retrieve information about solver state and results:

`vars` Returns a term "(X1,...,Xn) whose arity is the number of variables involved in the solver's constraint set, and whose arguments are these variables.

`ints` Returns a list [Xi1,...,Xik] which is the subset of the problem variables that the solver considers to be integers.

`constraints_norm` Returns a list of the problem constraints in normalised form. They may be simplified with respect to the original set that was passed to lp_setup/4.

`constraints` Returns a list of the problem constraints in denormalised (readable) form. They may be simplified with respect to the original set that was passed to lp_setup/4.

`objective` Returns a term min(E) or max(E), representing objective function and optimisation direction. E is a linear expression.

`method` Returns the method (`primal, dual, netprimal, netdual, barrier`) that is used to solve the problem.

`status` Status that was returned by the most recent invocation of the external solver.

`cost` Cost of the current solution. Fails if no solution has been computed yet.

`typed_solution` Returns a term "(X1,...,Xn) whose arguments are the properly typed (integer or float) solution values for the corresponding problem variables (`vars`). The floating point solutions are the same as returned by `solution`, the integers are obtained by rounding the corresponding floating-point solution to the nearest integer. To instantiate the problem variables to their solutions, unify this term with the corresponding term containing the variables:

```
instantiate_solution(Handle) :-
    lp_get(Handle, vars, Vars),
    lp_get(Handle, typed_solution, Values),
    Vars = Values.
```

`slack` Returns a list of floating-point values representing the constraint slacks. The order corresponds to the list order in `constraints`. Fails if no solution has been computed yet.

`dual_solution` Returns a list of floating-point values representing the dual solutions. The order corresponds to the list order in `constraints`. Fails if no solution has been computed yet.

`demon_tolerance` Returns a comma-separated pair `(RealTol,IntTol)` of floating-point values which specify how far outside a variable's range an lp-solution can fall before lp_demon_setup/6 re-triggers. The tolerances differ for real (default 0.00001) and integer (default 0.5) variables.

`simplex_iterations` Returns the external solver's count of simplex iterations.

`node_count` Returns the external MIP solver's node count.

`statistics` Returns a list of counter values `[Successes, Failures, Aborts]`, indicating how often lp_solve/2 was invoked on the Handle, and how many invocations succeeded, failed and aborted respectively.

Note that reduced_cost, slack, dual_solution can only be retrieved when previously requested in the option list of lp_setup/4 or with lp_set/3.

8.6.4 Accessing Variable-Related Information

Variable-related information can be retrieved individually for every variable without referring to a solver handle:

8.6.4.1 lp_var_get(+Var, +What, -Value)

Retrieve information about solver state and results related to a particular variable or constraint. Fails if no solution has been computed yet. What can take one of the following values:

`solution` Returns the floating-point solution for variable Var.

`typed_solution` Returns the properly typed (integer or float) solution for variable Var. For continuous variables, this is the same floating-point value as returned by `solution`, for integers the value is obtained by rounding the corresponding floating-point solution to the nearest integer.

`reduced_cost` Returns the reduced cost for variable Var.

Note that solution or reduced_cost can only be retrieved when previously requested in the option list of lp_setup/4 or with lp_set/3.

8.6.5 Collecting Linear Constraints

There are several ways to obtain a list of normalised constraints as input to lp_setup/4:

8.6.5.1 collect_lp_constraints_norm(-NormConstraints)

Collect all currently delayed linear constraints of the form X $= Y, X $>= Y or X $=< Y, and return them in normalised form. The corresponding delayed goals are removed (killed), but nonlinear constraints are ignored and remain delayed.

8.6.5.2 normalise_cstrs(+Constraints, -NormConstraints, -NonlinConstr)

where Constraints is a list of terms of the form X = Y, X >= Y or X =< Y (no dollar-signs!) where X and Y are arithmetic expressions. The linear constraints are returned in normalised form in NormConstraints, the nonlinear ones are returned unchanged in NonlinConstr.

8.6.5.3 Constraints from other solvers

For example, lib(fdplex) can extract the linear constraints from a set of finite-domain constraints.

8.6.6 Low-Level Interface Examples

8.6.6.1 Definition of optimize/2

The high-level predicate optimize/2 can be defined as:

```
optimize(OptExpr, ObjVal) :-
    collect_lp_constraints_norm(NormCstr),
    lp_setup(NormCstr, OptExpr, [integers(all)], Handle),
    lp_solve(Handle, ObjVal),
    lp_get(Handle, vars, VarVector),
    lp_get(Handle, typed_solution, SolutionVector),
```

```
VarVector = SolutionVector,
lp_cleanup(Handle).
```

First, all delayed goals of the form X $= Y, X $>= Y or X $=< Y are collected and normalised. Then a solver is set up, taking into account all integrality constraints. This solver is then invoked once, the solution vector obtained, and the variables instantiated to those solutions.

8.6.7 Access to Global Solver Parameters

The external Simplex solver has a number of global (i.e. not specific to a particular problem) parameters that affect the way it works. These can be queried and modified using the following predicates.

8.6.7.1 lp_get(optimizer, -Value)

Returns the name of the external optimizer, currently 'cplex' or 'xpress'.

8.6.7.2 lp_get(space, -Value)

This option only applies to solvers that require a-priory memory allocation (currently only XPRESS-MP). The value is a term of the form `space(Rows,Cols,NonZeros)` whose arguments are integers specifying how many extra rows, columns and nonzero coefficients can be added to a solver after it has been set up. The default is `space(0,0,0)`. It can be changed globally using lp_set/2 or on a per-solver basis using the space-option in lp_setup/4.

8.6.7.3 lp_get(+ParamName, -Value)

Retrieve the value of a global parameter for the external solver. The Value is either a float or an integer number, depending on the parameter. Refer to the solver documentation for details. The names of the parameters are as follows:

```
timelimit, time_limit, perturbation_const, lowerobj_limit, upperobj_limit,
feasibility_tol, markowitz_tol, optimality_tol, backtrack, treememory,
lowercutoff, uppercutoff, absmipgap, mipgap, integrality, objdifference,
relobjdifference, crash, dgradient, pricing, iisfind, netfind,
perturbation_ind, pgradient, refactor, iteration_limit, singularity_limit,
simplex_display, basisinterval, branch, cliques, covers, heuristic,
nodeselect, order, sosscan, startalgorithm, subalgorithm, variableselect,
solution_limit, node_limit, minsossize, mip_display, mip_interval, advance,
aggregator, coeffreduce, dependency, presolve, scale, xxxstart, reducecostfix
```

8.6.7.4 lp_set(+ParamName, +Value)

Set a global parameter for the external solver. The parameters are as in lp_get/2.

8.6.7.5 lp_set(space, +Value)

This setting only applies to solvers that require a-priory memory allocation (currently only XPRESS-MP). The value is a term of the form `space(+Rows,+Cols,+NonZeros)` whose arguments are integers specifying how many extra rows, columns and nonzero coefficients can

be added to a solver after it has been set up. The default is `space(0,0,0)`. The setting can be overwritten using the space-option in lp_setup/4.

8.6.7.6 int_tolerance(-Value)

The same as lp_get(integrality, Value): The solver's idea of an integer value, i.e. numbers within this tolerance from an integer are considered integers.

8.7 External Solver Output and Log

The external solver's output can be controlled using:

`lp_set(SolverChannel, +(Stream))` Send output from SolverChannel to the ECL^{i}PS^e I/O stream Stream.

`lp_set(SolverChannel, -(Stream))` Stop sending output from SolverChannel to the ECL^{i}PS^e I/O stream Stream.

SolverChannel is one of `result_channel, error_channel, warning_channel, log_channel`, and Stream is an ECL^{i}PS^e stream identifier (e.g. `output`, or the result of an open/3 operation). By default, `error_channel` and `warning_channel` are directed to ECL^{i}PS^e's `error` stream, while `result_channel` and `log_channel` are suppressed. To see the output on these channels, do

```
:- lp_set(result_channel, +output), lp_set(log_channel, +output).
```

Similarly, to create a log file:

```
:- open("mylog.log", write, logstream), lp_set(log_channel, +logstream).
```

and to stop logging:

```
:- lp_set(log_channel, -logstream), close(logstream).
```

8.8 Error Handling

If the external solver's optimization aborts with an error condition, or if limits are exceeded, the event 'CPLEX_ABORT' is raised. The default event handler is cplex_abort_handler/2, which prints a message and aborts. However, the handler is user-definable, so a more sophisticated handler could for instance change parameter settings and call lp_solve again.

Chapter 9

FDPLEX: A Hybrid Finite Domain / Simplex Solver

9.1 Motivation

Finite Domain Constraint Propagation and Integer Programming are two methods to solve and optimize systems of linear inequations over discrete domains. Experiments show that no one method has a general advantage over the other. Is seems rather that there are problems that are particularly well suited to either one or the other approach, owing to their different characteristics. But even different instances of the same problem can exhibit very different behaviours, which can make it impossible to chose the "most suitable" solver for a particular application.
These observations prompted the development of this library: It implements a hybrid solver based on cooperation between the ECL^{i}PS^e finite domain solver lib(fd) and the ECL^{i}PS^e/CPLEX interface lib(eplex). The basic idea is to have the programmable control provided within ECL^{i}PS^e, the incremental bound propagation achieved by the finite domain solver, and the global reasoning that is done by the simplex solver.

9.2 Usage

Many programs written for lib(fd) should run unchanged with lib(fdplex). The library is loaded using

```
:- lib(fdplex).
```

This will automatically load both the fd and the eplex library as well.
Note that this library is provided as source code. It really implements only one example of a solver cooperation. It is expected that users will modify the library to suit the special needs of the particular application.

9.3 Functionality

The library redefines minimize/2, min_max/2, indomain/1 and labeling/1 with versions that setup and trigger the simplex solver (on the relaxed floating-point problem) in appropriate places. In more detail:

1. At the beginning of minimize/2 or min_max/2, the finite-domain constraint store is scanned, an LP-relaxation is extracted, a corresponding LP-solver is set up and the relaxation is solved once.

2. Then the normal FD branch-and-bound procedure is started, using the user-supplied labeling routine.

3. The modified version of indomain/1 employs a value-selection strategy based on the solution of the LP-relaxation: The variable is first labeled with the integer which is closest to the floating-point solution. On backtracking, the rest of the domain is tried.

4. Variable instantiation (or, optionally, interval narrowing) can trigger the LP-solver: When a variable takes a value that is not close enough to the solution of the relaxation (or, optionally, when the narrowed interval excludes the solution of the relaxation), the solver is re-invoked. It computes a new solution, taking into account the current variable values and bounds.

The benefits from solver cooperation are:

- Infeasibility of the relaxed problem can prune the search.
- The cost of the relaxed solution is a lower bound to the cost of every integer solution. This cost bound is imposed as an additional constraint, and can thus cause FD-propagation and prune the search.
- The solution to the relaxed problem can be used as a labeling heuristics, hopefully leading to solution earlier.

9.4 FDPLEX Predicates

9.4.0.7 minimize(+Goal, +Expr) and min_max(+Goal, +Expr)

These are variants of the minimize/2 and min_max/2 predicates from the fd-library. They differ in that they set up a cooperating simplex solver prior to entering the branch-and-bound search.

9.4.0.8 indomain(+Var)

A variant of indomain/1 with modified value order: The integer that is closest to the relaxed-problem solution is chosen first.

9.4.0.9 labeling(+VarList)

A labeling routine using the modified indomain/1.

9.4.0.10 split_domain(+Var)

An alternative labeling primitive. It splits the variable's domain at the value suggested by the relaxed solution.

9.4.0.11 split_labeling(+VarList)

A labeling routine using split_domain/1 instead of indomain/1.

9.4.0.12 extract_lp_from_fd(-Constraints)

Extract a relaxed LP-problem from the finite-domain constraints (this is done implicitly in minimize and min_max). A list of =, >= and =< constraints is returned, and the variables involved are given the correct bounds and integer-type.

9.4.0.13 extract_lp_from_fd_norm(-NormConstraints)

Same as above, but the result is in normalised form, acceptable to lp_setup/4.

9.4.0.14 fdplex_statistics([Backtracks, SolverCalls, SolverFails, SolverBound])

Returns a list of counters giving information about the most recent invocation of minimize/min_max. Backtracks is the number of times indomain/1. has generated an alternative value. SolverCalls is the number of times the simplex solver was invoked. SolverFails counts how often the simplex detected infeasibility and SolverBound is the number of simplex solutions that were able to increase the lower cost bound. The difference SolverCalls-(SolverFails+SolverBound) represents the number of 'useless' simplex invocations, in the sense that these invocations didn't affect the search space. However, they might still have improved the labeling heuristics.

Chapter 10

REPAIR: Constraint-Based Repair

10.1 Introduction

The Repair library provides two simple, fundamental features which are the basis for the development of repair algorithms and non-monotonic search methods in ECL^{i}PS^e:

- The maintenance of *tentative values* for the problem variables. These tentative values may together form a partial or even inconsistent *tentative assignment*. Modifications to, or extensions of this assignment may be applied until a correct solution is found.
- The monitoring of constraints (the so called *repair constraints*) for being either satisfied or violated under the current tentative assignment. Search algorithms can then access the set of constraints that are violated at any point in the search, and perform repairs by changing the tentative assignment of the problem variables.

This functionality allows the implementation of classical local search methods within a CLP environment (see *Tutorial on Search Methods*). However, the central aim of the Repair library is to provide a framework for the integration of repair-based search with the consistency techniques available in ECL^{i}PS^e, such as the domains and constraints of the FD library.
A more detailed description of the theory and methods that are the basis of the Repair library is available [1].

10.1.1 Using the Library

To use the repair library you need to load it using

```
:- lib(repair).
```

Normally, you will also want to load one more of the 'fd', 'ria', 'range' or 'conjunto' solvers. This is because of the notion of tenability, i.e. whether a tentative value is in a domain is checked by communicating with a different solver that keeps that domain.

10.2 Tentative Values

10.2.1 Attaching and Retrieving Tentative Values

A problem variable may be associated with a tentative value. Typically this tentative value is used to record preferred or previous assignments to this variable.

10.2.1.1 ?Vars tent_set ++Values

Assigns tentative values for the variables in a term. These are typically used to register values the variables are given in a partial or initially inconsistent solution. These values may be changed through later calls to the same predicate. Vars can be a variable, a list of variables or any nonground term. Values must be a corresponding ground term. The tentative values of the variables in Vars are set to the corresponding ground values in Values.

10.2.1.2 ?Vars tent_get ?Values

Query the variable's tentative values. Values is a copy of the term Vars with the tentative values filled in place of the variables. If a variable has no tentative value a variable is returned in its place.

10.2.2 Tenability

A problem variable is *tenable* when it does not have a tentative value or when it has a tentative value that is consistent e.g. with its finite domain. For example

```
[eclipse 3]: X::1..5, X tent_set 3.
X = X{fd:[1..5], repair:3}
```

produces a tenable variable (note how the tentative value is printed as the variable's repair-attribute), while on the other hand

```
[eclipse 3]: X::1..5, X tent_set 7.
X = X{fd:[1..5], repair:7}
```

produces an untenable variable. Note that, unlike logical assignments, the tentative value can be changed:

```
[eclipse 3]: X::1..5, X tent_set 7, X tent_set 3.
X = X{fd:[1..5], repair:3}
```

10.2.2.1 tenable(?Var)

Succeeds if the given variable is tenable. This predicate is the link between repair and any underlying solver that maintains a domain for a variable[1].

10.2.3 The Tentative Assignment

The notion of a *tentative assignment* is the means of integration with the consistency methods of ECL^{i}PS^e. The tentative assignment is used for identifying whether a repair constraint is being violated.
The tentative assignment is a function of the groundness and tenability of problem variables according to the following table

[1] If you wish to write your own solver and have it cooperate with repair you have to define a test_unify handler

Variable Groundness	Variable Tenability	Value in Tentative Assignment
Ground	Tenable	Ground Value
Ground	Not Tenable	Ground Value
Not Ground	Tenable	Tentative Value
Not Ground	Not Tenable	Undefined

A repair constraint is violated under two conditions:

- The tentative assignment is undefined for any of its variables.
- The constraint fails under the tentative assignment.

10.2.4 Variables with No Tentative Value

It has been noted above that variables with no associated tentative value are considered to be tenable. Since no single value has been selected as a tentative value, the Repair library checks constraints for consistency with respect to the domain of that variable. A temporary variable with identical domains is substituted in the constraint check.

10.2.5 Unification

If two variables with distinct tentative values are unified only one is kept for the unified variable. Preference is given to a tentative value that would result in a tenable unified variable.

10.3 Repair Constraints

Once a constraint has been declared to be a repair constraint it is monitored for violation. Whether a repair constraint is considered to be violated depends on the states of its variables. A temporary assignment of the variables is used for checking constraints. This assignment is called the *tentative assignment* and is described above. A constraint which is violated in this way is called a *conflict constraint.*
Normal constraints are turned into repair constraints by giving them one of the following annotations:

10.3.0.1 Constraint r_conflict ConflictSet

This is the simplest form of annotation. It makes a constraint known to the repair library, i.e. it will initiate monitoring of **Constraint** for conflicts. When the constraint goes into conflict, it will show up in the conflict set denoted by **ConflictSet**, from where it can be retrieved using **conflict_constraints/2**. **Constraint** can be any goal that works logically, it should be useable as a ground check, and work on any instantiation pattern. Typically, it will be a constraint from some solver library. **ConflictSet** can be a user-defined name (an atom) or it can be a variable in which case the system returns a conflict set handle that can later be passed to **conflict_constraints/2**. Example constraint with annotation:

```
Capacity #>= sum(Weights)  r_conflict  cap_cstr
```

Note that using different conflict seta for different groups of constraints will often make the search algorithm easier and more efficient. A second allowed form of the **r_conflict** annotation is **Constraint r_conflict ConflictSet-ConflictData**. If this is used, **ConflictData** will appear in the conflict set instead of the **Constraint** itself. This feature can be used to pass additional information to the search algorithm.

10.3.0.2 Constraint r_conflict_prop ConflictSet

In addition to what **r_conflict** does, this annotation causes the **Constraint** to be activated as a goal as soon as it goes into conflict for the first time. If **Constraint** is a finite-domain constraint for example, this means that domain-based propagation on **Constraint** will start at that point in time.
Note that if you want constraint propagation from the very beginning, you should simply write the constraint twice, once without and once with annotation.

10.4 Conflict Sets

Given a tentative assignment, there are two kinds of conflicts that can occur:

- Untenable variables
- Violated constraints

To obtain a tentative assignment which is a solution to the given problem, both kinds of conflicts must be repaired. The repair library supports this task by dynamically maintaining conflict sets. Typically, a search algorithm examines the conflict set(s) and attempts to repair the tentative assignment such that the conflicts disappear. When all conflict sets are empty, a solution is found.

10.4.0.3 conflict_vars(-Vars)

When a variable becomes untenable, it appears in the set of conflict variable, when it becomes tenable, is disappears. This primitive returns the list of all currently untenable variables. Note that all these variables must be reassigned in any solution (there is no other way to repair untenability). Variable reassignment can be achieved by changing the variable's tentative value with tent_set/2, or by instantiating the variable. Care should be taken whilst implementing repairs through tentative value changes since this is a non-monotonic operation: conflicting repairs may lead to cycles and the computation may not terminate.

10.4.0.4 conflict_constraints(+ConflictSet, -Constraints)

When a repair constraint goes into conflict (i.e. when it does not satisfy the tentative assignment of its variables), it appears in a conflict set, once it satisfies the tentative assignment, it disappears. This primitive returns the list of all current conflict constraints in the given conflict set. **ConflictSet** is the conflict set name (or handle) which has been used in the corresponding constraint annotation. For example

```
conflict_constraints(cap_cstr, Conflicts)
```

would retrieve all constraints that were annotated with `cap_cstr` and are currently in conflict. At least one variable within a conflict constraint must be reassigned to get a repaired solution. Variable reassignment can be achieved by changing the variable's tentative value with tent_set/2, or by instantiating the variable. Care should be taken whilst implementing repairs through tentative value changes since this is a non-monotonic operation: conflicting repairs may lead to cycles and the computation may not terminate.
Note that any repair action can change the conflict set, therefore **conflict_constraints/2** should be called again after a change has been made, in order to obtain an up-to-date conflict set.

10.4.0.5 poss_conflict_vars(+ConflictSet, -Vars)

The set of variables within the conflict constraints. This is generally a mixture of tenable and untenable variables.

10.5 Invariants

For writing sophisticated search algorithms it is useful to be able not only to detect conflicts caused by tentative value changes, but also to compute consequences of these changes. For example, it is possible to repair certain constraints automatically by (re)computing one or more of their variable's tentative values based on the others (e.g. a sum constraint can be repaired by updating the tentative value of the sum variable whenever the tentative value of one of the other variables changes). We provide two predicates for this purpose:

10.5.0.6 -Result tent_is +Expression

This is similar to the normal arithmetic **is/2** predicate, but evaluates the expression based on the tentative assignment of its variables. The result is delivered as (an update to) the tentative value of the Result variable. Once initiated, **tent_is** will stay active and keep updating Result's tentative value eagerly whenever the tentative assignment of any variable in Expression changes.

10.5.0.7 tent_call(In, Out, Goal)

This is a completely general meta-predicate to support computations with tentative values. Goal is a general goal, and In and Out are lists (or other terms) containing subsets of Goal's variables. A copy of Goal is called, with the In-variables replaced by their tentative values and the Out-variables replaced by fresh variables. Goal is expected to return values for the Out variables. These values are then used to update the tentative values of the original Out variables. This process repeats whenever the tentative value of any In-variable changes.

10.5.0.8 Waking on Tentative Assignment Change

The predicates **tent_is/2** and **tent_call/3** are implemented using the **ga_chg** suspension list which is attached to every repair variable. The programmer has therefore all the tools to write specialised, efficient versions of tent_call/3. Follow the following pattern:

```
my_invariant(In, Out) :-
        In tent_get TentIn,
        ... compute TentOut from TentIn ...
        suspend(my_invariant(In,Out,Susp), 3, [In->ga_chg]),
        Out tent_set TentOut.
```

This can be made more efficient by using a demon (**demon/1**).

10.6 Examples

More examples of repair library use, in particular in the area of local search, can be found in the *Tutorial on Search Methods*.

10.6.1 Interaction with Propagation

In the following example, we set up three constraints as both repair and fd-constraints (using the **r_conflict_prop** annotation) and install an initial tentative assignment (using **tent_set**). We then observe the result by retrieving the conflict sets:

```
[eclipse 1]: lib(repair), lib(fd).              % libraries needed here
yes.
[eclipse 2]:
        [X,Y,Z]::1..3,                          % the problem variables
        Y #\= X r_conflict_prop confset,        % state the constraints
        Y #\= Z r_conflict_prop confset,
        Y #=  3 r_conflict_prop confset,
        [X,Y,Z] tent_set [1,2,3],               % set initial assignment
        [X,Y,Z] tent_get [NewX,NewY,NewZ],      % get repaired solution
        conflict_constraints(confset, Cs),      % see the conflicts
        conflict_vars(Vs).

X = X{fd:[1..3], repair:1}
Y = 3
Z = Z{fd:[1, 2], repair:3}
NewX = 1
NewY = 3
NewZ = 3
Cs = [3 #\= Z{fd:[1, 2], repair:3}]
Vs = [Z{fd:[1, 2], repair:3}]

Delayed goals:
     ...
yes.
```

Initially only the third constraint `Y #= 3` is inconsistent with the tentative assignment. According to the definition of **r_conflict_prop** this leads to the constraint `Y #= 3` being propagated, which causes Y to be intantiated to 3 thus rendering the tentative value (2) irrelevant.

Now the constraint `Y #\= Z`, is in conflict since Y is now 3 and Z has the tentative value 3 as well. The constraint starts to propagate and removes 3 from the domain of `Z [1..2]`.
As a result Z becomes a conflict variable since its tentative value (3) is no longer in its domain.
The `Y #\= Z` constraint remains in the conflict constraint set because Z has no valid tentative assignment.
The constraint `Y #\= X` is not affected, it neither goes into conflict nor is its fd-version ever activated.
To repair the remaining conflicts and to find actual solutions, the `repair/0` predicate described below could be used.

10.6.2 Repair Labeling

This is an example for how to use the information provided by the repair library to improve finite domain labeling. You can find the repair/0 predicate in the 'repairfd' library file.

```
repair :-
    ( conflict_vars([C|_]) ->       % label conflict
        indomain(C),                %  variables first
        repair
    ; conflict_constraints([C|_]) ->
        term_variables(C, Vars),    % choose one variable in
        deleteffc(Var,Vars, _),     %  the conflict constraint
        Var tent_get Val,
        (Var = Val ; Var #\= Val),
        repair
    ;                               % no more conflicts:
        true                        % a solution is found.
    ).
```

The predicate is recursive and terminates when there are no more variables or constraints in conflict.
Repair search often finishes without labeling all variables, a solution has been found and a set of tenable variables are still uninstantiated. Thus even after the search is finished, Repair library delayed goals used for monitoring constraints will be present in anticipation of further changes.
To remove them one has to ground these tenable variables to their tentative values.
Note that the example code never changes tentative values. This has the advantage that this is still a complete, monotonic and cycle-free algorithm. However, it is not very realistic when the problem is difficult and the solution is not close enough to the initial tentative assignment. In that case, one would like to exploit the observation that it is often possible to repair some conflict constraints by changing tentative values. During search one would update the tentative values to be as near as pssible to what one wants while maintaining consistency. If the search leads to a failure these changes are of course undone.

Chapter 11

RIA: ECL^{i}PS^e Real Number Interval Arithmetic

11.1 Introduction

11.1.1 What Ria does

The Ria library solves constraint problems over the reals. It is not limited to linear constraints. So it can be used to solve general problems like:

```
[eclipse 2]: ln(X) *>= sin(X).

X = X{0.36787944117144233 .. Infinity}
yes.
```

The Ria library has two different algorithms built in. The default one is arc-consistency and is quite cheap, the other provides a stronger consistency but is slower.
Both algorithms work on the same data representation. That is real numbers in a closed range between (and including) two floats. The library will reduce this range if possible. It never gets as far as reducing a variable to a single float.

11.1.2 Usage

Load the library by using

```
:- lib(ria).
```

You will need ECL^{i}PS^e version 3.5.2 or higher.
Note that version 3.5.2 does not treat floating point infinities properly, in particular, they are printed in a strange way and the normal arithmetic predicates like **is/2** cannot cope with them. However, the ria-library still works fine. Use `inf` to denote infinity in version 3.5.2. Later versions of ECL^{i}PS^e fully support computation with infinities and allow the syntax `[+-]1.0Inf`.

11.1.3 History

This work was triggered by the work of Slava Zilberfaine on an interface between ECL^{i}PS^e and Unicalc. Several shortcomings of this interface prompted us to develop lib(ria), which does not share any code with Unicalc.

11.2 Library Predicates

11.2.1 Ranged and Typed Variables

Vars :: Lo..Hi Logically: Constrain a variable (or all variables in a list) to take values between and including Lo and Hi. The type of the bounds determines the type of the variable (real or integer). It is possible to use the bounds `-inf` (or `-1.0Inf`) and `-inf` (or `1.0Inf`) to represent infinities. This is the default range used for variables where no range has been declared.

Operationally: This information is immediately stored into the variable's attribute. The bounds are also widened by one float below and above to ensure the bounds are included in the range.

reals(Vars) Equivalent to `Vars :: -inf..inf`

integers(Vars) The given variables can only take integer values.

11.2.2 Constraints

ExprX *= ExprY ExprX is equal to ExprY. ExprX and ExprY are general expressions.

ExprX *>= ExprY ExprX is greater or equal to ExprY. ExprX and ExprY are general expressions.

ExprX *=< ExprY ExprX is less or equal to ExprY. ExprX and ExprY are general expressions.

Var iis SimpleExpr This is the simple, uni-directional constraint that is used by the solver to rewrite all other constraints. It is not meant for use inside a program, but it shows up among the delayed goals.

11.2.3 Arithmetic Expressions

The following arithmetic expression can be used inside the constraints:

X Variables. If X is not yet a ranged variable, it is turned into one via an implicit declaration `X :: -inf..inf`.

123 Integer constants. They are assumed to be exact and are used as is.

0.1 Floating point constants. They are assumed to be inexact and are widened into a narrow interval that is guaranteed to contain the true value.

exact(0.5) Sometimes the programmer knows that a floating point constant is exact or meant to be taken literally. In that case, use this form.

pi, e Intervals enclosing the constants pi and e respectively.

inf Floating point infinity.

+Expr Identity.

-Expr Sign change.

+-Expr Expr or -Expr. The result is an interval enclosing both.

abs(Expr) The absolute value of Expr.

E1+E2 Addition.

E1-E2 Subtraction.

E1*E2 Multiplication.

E1/E2 Division.

E1 ∧E2 Exponentiation.

min(E1,E2) Minimum.

max(E1,E2) Maximum.

sqr(Expr) Square. Logically equivalent to Expr*Expr, but with better operational behaviour.

sqrt(Expr) Square root (always positive).

exp(Expr) Same as e ∧Expr.

ln(Expr) Natural logarithm, the reverse of the exp function.

sin(Expr) Sine.

cos(Expr) Cosine.

atan(Expr) Arcus tangens.

rsqr(Expr) Reverse of the sqr function. The same as +-sqrt(Expr).

rpow(Expr,N) Reverse of Expr ∧N, where N is an integer constant.

(E1;E2) E1 or E2. Operationally, this computes the union of two intervals.

sub(Expr) A subinterval of Expr.

11.2.4 Solving by Interval Propagation

Some problems can be solved just by interval propagation, for example:

```
[eclipse 9]: X :: 0.0..100.0, sqr(X) *= 7-X.

X = X{2.1925824014821349 .. 2.1925824127108311}

Delayed goals:
...
yes.
```

There are two things to note here:

- The solver never instantiates real-variables. They only get reduced to narrow ranges.
- In general, many delayed goals remain at the end of propagation. This reflects the fact that the variable's ranges could possibly be further reduced later on during the computation. It also reflects he fact that
- the solver does not guarantee the existence of solutions in the computed ranges. However, it guarantees that there are no solutions outside these ranges.

Note that, since variables by default range from minus to plus infinity, we could have written the above example as:

```
[eclipse 2]: sqr(X) *= 7-X, X *>= 0.

X = X{2.1925824014821349 .. 2.1925824127108311}

Delayed goals:
...
yes.
```

If too little information is given, the interval propagation may not be able to infer any interesting bounds:

```
[eclipse 2]: sqr(X) *= 7-X.

X = X{-1.0Inf .. 7.0000000000000009}

Delayed goals:
...
yes.
```

11.2.5 Reducing Ranges Further

There are two methods for further domain reduction. They both rely on search and splitting the domains. There are 2 parameters to specify how domains are to be split.
The *Precision* parameter is used to specify the minimum required precision, i.e. the maximum size of the resulting intervals. Note that the arc-propagation threshold needs to be one or

several orders of magnitude smaller than *precision*, otherwise the solver may not be able to achieve the required precision.
The *lin/log* parameter guides the way domains are split. If it is set to *lin* then the split is in the arithmetic middle. If it is set to *log*, the split is such as to have the same number of floats to either side of the split. This is to take the logarithmic distribution of the floats into account.
If the ranges of variables at the start of the squashing algorithm are known not to span several orders of magnitude ($|max| < 10 * |min|$) the somewhat cheaper linear splitting may be used. In general, log splitting is recommended.

locate(+Vars, +Precision)

locate(+Vars, +Precision, +lin/log) Locate solution intervals for the given variables with the required precision. This works well if the problem has a finite number of solutions. locate/2,3 work by nondeterministically splitting the ranges of the variables until they are narrower than Precision.

squash(+Vars, +Precision, +lin/log) Use the squash algorithm (section 11.3.3) on these variables. This is a deterministic reduction of the ranges of variables, done by searching for domain restrictions which cause failure, and then reducing the domain to the complement of that which caused the failure. This algorithm is appropriate when the problem has continuous solution ranges (where locate would return many adjacent solutions).

locate(+LocateVars,+SquashVars,+Precision,+lin/log) A variant of locate/2,3 with interleaved squashing: The squash algorithm (section 11.3.3) is once applied to the SquashVars initially, and then again after each splitting step, ie. each time one of the LocateVars has been split nondeterministically. A variable may occur both in LocateVars and SquashVars.

11.2.6 Setting the Arc-Propagation Threshold

Limiting the amount of propagation is important for efficiency. A higher threshold speeds up computations, but reduces precision and may in the extreme case prevent the system from being able to locate individual solutions.

set_threshold(+Threshold) Set the threshold to Threshold which is a small floating-point number. This means any propagation which results in a domain reduction smaller than Threshold will not be executed. The default is 1e-8.

get_threshold(-Threshold) Read the current threshold.

11.2.7 Obtaining Solver Statistics

Often it is difficult to know where the solver spends its time. The library has built-in counters which keep track of

- Propagation steps (**prop**)
- Domain splits in locate/2,3,4 (**split**)
- Attempts to bound reduction in squash/3 or locate/4 (**squash**)

The counters are controlled using the primitive

ria_stat(on)

ria_stat(off) Enables/disable collection of statistics. Default is off.

ria_stat(reset) Reset statistics counters.

ria_stat(-Stat) Returns a list of CounterName=CounterValue pairs, summarising the computation since the last reset.

ria_stat(print) Print statistics counters.

11.3 The Ria library algorithms

11.3.1 Arc consistency

Ria uses an arc consistency propagation algorithm. This terminates when all arcs are consistent, i.e. when for each variable, setting it to a value outside it's range would violate at least one constraint.
In a preprocessing step complex constraints are broken up into simple directed constraints. If necessary, auxiliary variables are introduced. For example:

```
X * (Y+Z) = 1
```

rewrites into

```
Aux  iis  Y + Z,
Z    iis  Aux - Y,
Y    iis  Aux - Z,
X    iis  1 / Aux,
Aux  iis  1 / X
```

Changes in the ranges of the input variables (right hand side) trigger the constraints to recompute the range for the output variable (left hand side).
At any time several constraints may be triggered. A heuristic favouring constraints that have been succesful at trimming variable ranges in the past is used for selection of the next constraint to compute.

11.3.2 Arc consistency threshold

If the execution of a constraint, restricts the range of a variable by a quantity less than the propagation threshold, this restriction is simply not applied. This terminates propagation early and prevents almost infinite loops of ever tinier propagations on ill-behaved problems. For example:

```
[eclipse 17]: set_threshold(1e-3).

yes.
[eclipse 18]: sin(X) *= X.
```

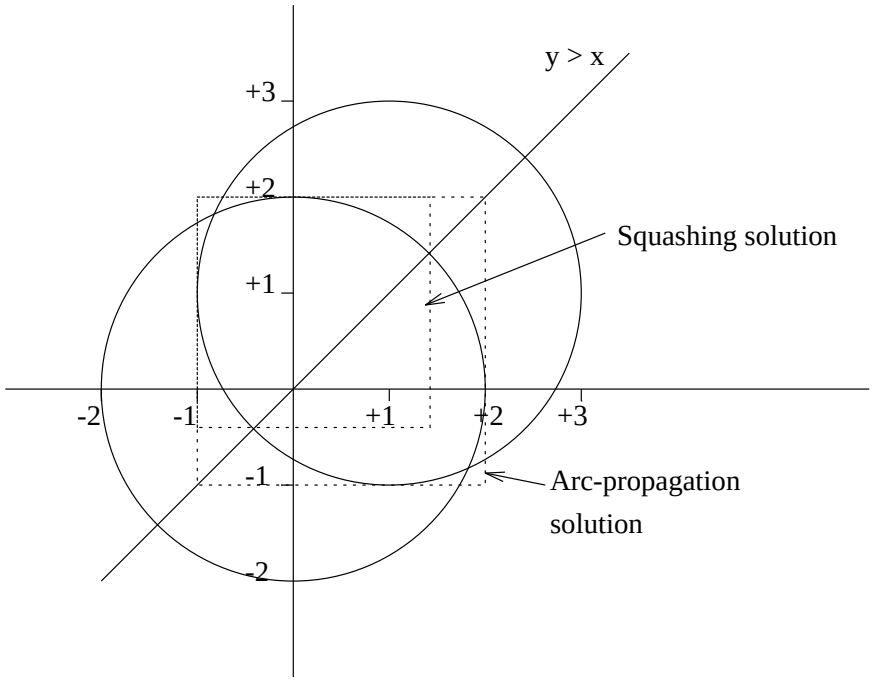

Figure 11.1: Propagation with Squash algorithm (example)

```
X = X{-0.18143335721992979 .. 0.18143335721992984}
yes.
[eclipse 19]: Y is 0.18143335721992981 - sin(0.18143335721992981).

Y = 0.00099376872589851394
yes.
[eclipse 20]:
```

For small X X and sin(X) are almost identical, the library in this case will have made two variables and two directed constraints out of the above example.
These are:

```
sin(X) -> S
arcsin(S) -> X
```

Since each propagation only makes tiny differences to the domains of S and X the algorithm stops.
Intuitively this slowly convergent behaviour happens when the solution is a point where two curves meet at a tangent.

11.3.3 Squash algorithm

A stronger propagation algorithm is also included. This is built upon the normal arc consistency. It guarantees that, if you take any variable and restrict its range to a small domain near one of its bounds, the original arc consistency solver will not find any constraint unsatisfied. All points (X,Y) Y >= X, lying within the intersection of 2 circles with radius 2, one centred at (0,0) the other at (1,1).

```
[eclipse 29]: 4 *>= X^2 + Y^2, 4 *>= (X-1)^2+(Y-1)^2, Y *>= X.

Y = Y{-1.0000000000000016 .. 2.0000000000000013}
X = X{-1.0000000000000016 .. 2.0000000000000013}
yes.
[eclipse 30]:
```

The arc-consistency solution does not take into account the X >= Y constraint. Intuitively this is because it passes through the corners of the box denoting the solution to the problem of simply intersecting the two circles.

```
[eclipse 29]: 4 *>= X^2 + Y^2, 4 *>= (X-1)^2+(Y-1)^2, Y *>= X,
squash([X,Y],1e-5,lin).

X = X{-1.0000000000000016 .. 1.4142135999632603}
Y = Y{-0.41421359996326074 .. 2.0000000000000013}
yes.
```

Index

Bibliography

[1] Hani El Sakkout. *Improving Backtrack Search: Three Case Studies of Localized Dynamic Hybridization.* PhD thesis, Imperial College, London, June 1999.

[2] T. Fruehwirth. Constraint simplification rules. Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992. presented at CLP workshop at ICLP 92, Washington, USA, November 1992.

[3] T. Fruehwirth. Temporal reasoning with constraint handling rules. Technical Report Core-93-8, ECRC Munich, Germany, January 1993.

[4] T. Fruehwirth and Ph. Hanschke. Terminological reasoning with constraint handling rules. In *First Workshop on the Principles and Practice of Constraint Programming*, Newport, RI, USA, April 1993.

[5] T. Le Provost. Approximate Generalised Propagation. ESPRIT Project Deliverable CORE-93-7, also as CHIC-WP5-D.5.2.3.3, ECRC GmbH, January 1993.

[6] T. Le Provost and M. Wallace. Domain-independent propagation (or Generalised Propagation). In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'92)*, pages 1004–1011, June 1992.

[7] T. Le Provost and M. Wallace. Constraint satisfaction over the CLP Scheme. *Journal of Logic Programming*, 16(3-4):319–359, July 1993. Special Issue on Constraint Logic Programming.