ECL^{i}PS^e Constraint Library Manual

Release 5.8

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

January 25, 2005

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.
In this section we shall briefly summarize the constraint solvers that are available as ECL^{i}PS^e libraries. No examples are given here - each solver has its own documentation with examples for the interested reader.

1.1 Suspended Goals: *suspend*

The constraint solvers of ECL^{i}PS^e are all implemented using suspended goals. In fact the simplest implementation of any constraint is to suspend it until all its variables are sufficiently instantiated, and then test it.
The library *suspend* contains versions of the mathematical constraints `>=`, `>`, `=:=`, `=\=`, `=<`, `<` which behave like this[1].

1.2 Finite Domains: *ic*

1.2.1 *Integer Domain*

The standard constraint solver offered by most constraint programming systems is the *finite domain* solver, which applies constraint propagation techniques developed in the AI community [21]. ECL^{i}PS^e supports finite domain constraints via the *ic* library[2]. This library implements finite domains of integers, and the usual functions and constraints on variables over these domains.

[1] Note that the global flag *coroutine* has a similar effect: it causes the arithmetic comparisons as well as many other built-in predicates to delay until they are sufficiently instantiated

[2] There is also an older implementation, the *fd* library, whose use is deprecated

1.2.2 Symbolic Domain: *ic_symbolic*

In addition to integer domains, ECL^{i}PS^e offers finite domains of ordered non-numeric values, for example $red, green, blue$. These are implemented by the *ic_symbolic* library.
Whilst there is a standard set of constraints supported by the *ic* library in ECL^{i}PS^e and in most constraint programming systems, many more finite domain constraints have been introduced which have uses in specific applications and do not belong in a generic constraint programming library. The behaviour of these constraints is to prune the finite domains of their variables, in just the same way as the standard constraints. Therefore ECL^{i}PS^e offers several further libraries which implement more constraints using the *ic* library.

1.2.3 Global Constraints: *ic_global*

One such library is *ic_global*. It supports a variety of constraints, each of which takes as an argument a list of finite domain variables, of unspecified length. Such constraints are called "global" constraints [2]. Examples of such constraints, available from the ic_global library are `alldifferent/1`, `maxlist/2`, `occurrences/3` and `sorted/2`.

1.2.4 Scheduling Constraints

There are several ECL^{i}PS^e libraries implementing global constraints for scheduling applications. The constraints have the same semantics, but different propagation. The constraints take a list of tasks (start times, durations and resource needs), and a maximum resource level. They reduce the finite domains of the task start times by reasoning on resource bottlenecks [13]. Three ECL^{i}PS^e libraries implementing scheduling constraints are *cumulative*, *edge_finder* and *edge_finder3*.

1.3 Sets

ECL^{i}PS^e offers constraint solving over the domain of finite sets of integers. The *ic_sets* library works together with the *ic* library to reason about sets and set cardinality [10][3].

1.4 Intervals

Besides finite domains, ECL^{i}PS^e also offers continuous domains in the form of numeric intervals. These are also implemented by the *ic* library, which is an integration of an integer finite domain solver and interval reasoning over continuous intervals[4]. It solves equations and inequations between general arithmetic expressions over continuous or integral variables. The expressions can include non-linear functions such as *sin*, built-in constants such as *pi*. Piecewise linear unary functions are also available.
In addition to constraints, *ic* offers search techniques (*splitting* [20] and *squashing* [17]) for solving problems involving continuous numeric variables.

[3]There is also an older implementation, the *conjunto* library, which is generally less efficient, but implements sets of symbolic elements as well as integer sets

[4]The *ic* library replaces the old *ria* interval solver, and covers most of the functionality of the finite domain solver *fd*

1.5 User-Defined Constraints

1.5.1 Generalised Propagation: *propia*

The predicate *infers* takes as one argument any user-defined predicate, and as a second argument a form of propagation to be applied to that predicate.
This functionality enables the user to turn any predicate into a constraint [16]. The forms of propagation include finite domains and intervals.

1.5.2 Constraint Handling Rules

The user can also specify predicates using rules with guards [9]. They delay until the guard is entailed or disentailed, and then execute or terminate accordingly.
This functionality enables the user to implement constraints in a way that is clearer than directly using the underlying *suspend* library.

1.6 Repair

The *repair* library allows a *tentative* value to be associated with any variable [22]. This tentative value may violate constraints on the variable, in which case the constraint is recorded in a list of violated constraints. The repair library also supports propagation *invariants* [18]. Using invariants, if a variable's tentative value is changed, the consequences of this change can be propagated to any variables whose tentative values depend on the changed one. The use of tentative values in search is illustrated in the ECL^{i}PS^e "Tutorial on Search Methods".

1.7 Linear Constraints

There are two libraries supporting linear constraint solving. The first *eplex* provides an interface to external linear programming packages. It offers flexibility and scalability, but may require a license for the external software. The second *clpqr* can support infinite precision, but is less efficient and scalable and offers fewer facilities.

1.7.1 External Linear Solvers: *eplex*

eplex supports a tight integration [3] between external linear solvers (CPLEX [12] and XPRESS [19]) and ECL^{i}PS^e. Constraints as well as variables can appear in both the external linear solver and other ECL^{i}PS^e solvers. Variable bounds are automatically passed from the ECL^{i}PS^e *range* solver to the external solver. Optimal solutions and other solutions can be returned to ECL^{i}PS^e as required. Search can be carried out either in ECL^{i}PS^e or in the external solver.

1.7.2 *clpqr*

The *clpqr* library offers two implementations of the Simplex method for solving linear constraints [11]. One version uses rationals and is exact. The other version uses floats. This library employs public domain software, and can be used for small problems (with less than 100 variables).

1.7.3 Piecewise Linear: *eplex_relax*

This library handles any user-defined piecewise linear function as a constraint closely integrated with *eplex*. It offers better pruning than the standard handling of piecewise linear constraints in the external solvers [1].

1.7.4 Probing for Scheduling

For scheduling applications where the cost is dependent on each start time, a combination of solvers can be very powerful. For example, we can use finite domain propagation to reason on resources and linear constraint solving to reason on cost [4].
The *probing_for_scheduling* library supports such a combination, via a similar user interface to the *cumulative* constraint mentioned above.

1.8 Other Libraries

The solvers described above are just a few of the many libraries available in ECLiPSe and listed in the ECL^{i}PS^e library directory. Any ECL^{i}PS^e user who has implemented a constraint solver is welcome to send the code to the ECL^{i}PS^e team at IC-Parc so that it can be added to the available libraries. Comments and suggestions on the existing libraries are also welcome!

Chapter 2

Common Solver Interface

2.1 Introduction

ECL^{i}PS^e now provides a common syntax for the main arithmetic constraints provided by different constraint solvers. The basic idea is that the name and syntax of the constraint determines the declarative meaning, while the operational semantics (the algorithmic constraint behaviour) is determined by the module which implements the constraint. This principle simplifies the development of applications that use hybrid solution methods. Constraints can be passed easily to different, even multiple, solvers.

2.2 Common constraints

The constraints can be divided into the following groups:

- the numeric type constraints reals/1 and integers/1. Note that in this context, integers are considered a subset of the reals.
- the range constraints ::/2, `#::` and $::/2, which give upper and lower bounds to their variables. In addition, ::/2 and `#::` can also imply integrality.
- arithmetic equality, inequality and disequality over the mathematical real numbers, e.g. `$=`, `$>=`, `>`, `$\=` (and their symnonyms `=:=`, `>=`, `>`, `=\=`). Note that in this context, integers are considered a subset of the reals and can therefore occur in these constraints.
- arithmetic equality, inequality and disequality which in addition to the above constrain all variables within their arguments to integers. Syntactically, these generally have a leading `#`, e.g. `#=`, `#\=`, `#<`.

Not all constraints are supported by all the solvers. For example, the finite domain solver does not support any of the constraints that take real numbers. Table 2.1 shows the constraints that are available from the various constraint solvers. In the table, a 'yes' entry indicates that the particular constraint is supported by the particular solver. Note that some further restrictions may apply for a particular solver. For example, the finite domain solver can handle only linear expressions. Refer to the documentation for each individual solver to see what restrictions might apply.

	$::2 $=/2 $>=/2 $=</2 =:=/2 >=/2 =</2	$>/2 $</2 >/2 </2	$\=/2 =\=/2	::/2	#::/2 #=/2 #>=/2 #=</2 #>/2 #</2	#\=/2	integers/1	reals/1
suspend	yes	yes	yes	yes	yes	yes	yes	yes
ic	yes	yes	yes	yes	yes	yes	yes	yes
eplex	yes	—	—	yes	—	—	yes	yes
colgen	yes	—	—	yes	—	—	—	yes
(arith)	(yes)	(yes)	(yes)	—	—	—	—	—

- If integer bounds are given to the eplex version of `::/2` the external solver does not consider this as an integrality constraint and only solves the continuous relaxation which can then be rounded to the next integer. To make the external solver solve a mixed integer problem, use the eplex version of integers/1.

Table 2.1: Supported constraints for various arithmetic solvers

Note that the last line, labelled 'arith', is not really a constraint solver but represents just the standard arithmetic tests which require all variables to be instantiated. This behaviour is provided by the (automatically imported) module `eclipse_language`.
It can be somewhat confusing that these standard arithmetic tests have the same names as the corresponding constraints. One one hand, they have the same declarative meaning. On the other hand, they are not really interchangeable because they can only be used as tests, not as active constraints. The following synonyms are therefore provided to make the distinction visible where needed, and to reduce the need for module-qualification:

$=/2	=:=/2
$\=/2	=\=/2
$>=/2	>=/2
$=</2	=</2
$>/2	>/2
$</2	</2

2.3 Using the constraints

To use the constraints, ECL^{i}PS^e needs to know which solver to pass a particular constraint to. The easiest method for doing this is to module qualify the constraint. For example,

```
..., ic: (A #>= B), ...
```

passes the constraint `A #>= B` to the ic solver. The solver must be loaded first (e.g. via lib/1) before any constraint can be passed to it.
A constraint can also be passed to more than one solver by specifying a list in the module qualification. For example,

```
..., [ic, suspend]: (A #>= B), ...
```

will pass the constraint to both the ic and suspend solvers.
Module qualification is *not* needed if the constraint is defined by an imported module, and there is no other imported module which defines the same constraint. For example, if ic is the only imported module which defines `#>=/2`, then `#>=/2` can be used without qualification:

```
..., A #>= B, ...
```

Note that for constraints that are defined for `eclipse_language`, such as `>=` (the standard arithmetic test), the default behaviour when an unqualified call to such a constraint is made is to pass it to `eclipse_language`, even if another solver which defines the constraint is imported. Thus, for example

```
..., A >= B, ...
```

will by default have standard (i.e. non-suspending) test semantics, even if, e.g. the `ic` library (which also defines `>=/2`) is imported. To access the `ic` version, module qualification should be added:

```
..., ic:(A >= B), ...
```

Alternatively, the synonymous `$>=/2` constraint could be used:

```
..., A $>= B, ...
```

In general, module qualifications are recommended if the programmer wants to ensure a particular constraint behaviour regardless of which other modules might be loaded. On the other hand, if the intention is to switch easily between different solvers by simply loading a different library, module qualification is best omitted.
Finally, it is also possible to let the running program determine which solver to use. In this case, the program has a variable in the module position, which will only be bound at runtime:

```
..., Solver:(A #>= B), ...
```

This will however prevent the solver from performing any compile-time preprocessing on the constraint.

2.4 The Solvers

suspend This is the simplest possible 'solver'. Its behaviour is to wait until all variables in a constraint have been instantiated to numbers. Then it performs a test to check whether the constraint is satisfied, and fails if this is not the case.

ic A new hybrid solver, combining integer and real interval constraint solving. This solver is intended to replace the FD (and the already discontinued RIA) solver. For more information, please see chapter 3.

eplex An interface to an LP or MIP solver, i.e. it implements linear constraints over reals or integers.

arith This is not really a solver, but just the implementation of simple arithmetic tests in module `eclipse_language`. These require that all variables are instantiated when the test is invoked. The reason to list it here is that the proper solvers use the same syntax and can be considered generalisations of the traditional tests.

Chapter 3

IC: A Hybrid Finite Domain / Real Number Interval Constraint Solver

3.1 Introduction

The IC (Interval Constraint) library is a new hybrid integer/real interval arithmetic constraint solver. Its aim is to make it convenient for programmers to write hybrid solutions to problems, mixing together integer and real constraints and variables.
Previously, if one wished to mix integer and real constraints, one had to import separate solvers for each, with the solvers using different representations for the domains of variables. This meant any variable appearing in both kinds of constraints would end up with two domain representations, with extra constraints necessary to keep the two representations synchronised.

3.1.1 What IC does

IC is a general interval propagation solver which can be used to solve problems over both integer and real variables. Integer variables behave much like those from the old finite domain solver FD, while real variables behave much like those from the old real interval arithmetic solver RIA. IC allows both kinds of variables to be mixed seamlessly in constraints, since (with a few exceptions) the same propagation algorithms are used throughout and all variables have the same underlying representation (indeed, a real variable can be turned into an integer variable simply by imposing an integrality constraint on it).
IC replaces the 'fd', 'ria' and 'range' libraries. Since IC does not support symbolic domains, there is a separate symbolic solver library 'ic_symbolic', to provide the non-numeric functionality of 'fd'.

3.1.2 Differences between IC and FD

- IC supports real variables and constraints; FD does not.
- FD supports symbolic domains; IC does not (use the ic_symbolic library).
- In FD, numeric domains are more or less limited to -10000000..10000000 (this default domain can be modified, but the larger one makes it, the more likely one is to run into machine integer overflow problems). In IC there is no limit as such, and bounds on integer variables can be infinite (though variables should not be assigned infinite values).

However, since floating point numbers are used in the underlying implementation, not every integer value is representable. Specifically, integer variables and constraints ought to behave as expected until the values being manipulated become large enough that they approach the precision limit of a double precision floating point number (2^{51} or so). Beyond this, lack of precision may mean that the solver cannot be sure which integer is intended, in which case the solver starts behaving more like an interval solver than a traditional finite domain solver. Note however that this precision limit is way beyond what is normally supported by finite domain solvers (typically substantially less than 2^{32}). Note also that deliberately working with integer variables in this kind of range is not particularly recommended; the main intention is for the computation to be "safe" if it strays up into this region by ensuring no overflow problems.

- IC usually requires that expressions constructed at runtime be wrapped in **eval/1** when they appear in constraints; otherwise the variable representing the express may be assumed to be an IC variable, resulting in a type error. See section 3.1.7 for more details. We hope to remove this limitation in a future release.

- IC does not support the **#<=/2** syntax for less-than-or-equal constraints. Use **#=</2** (the standard ECL^{i}PS^e operator for integer less-than-or-equal constraints, also supported by FD) instead. Similarly, use **#\=/2** instead of **##/2**.

- The reified connectives provided by the two solvers are different: FD's **#\+/1**, **#/\/2**, **#\//2**, **#=>/2** and **#<=>/2** (and their reified versions) correspond to IC's **neg/1**, **and/2**, **or/2**, **=>/2** and **#=/2** (and their reified versions). Note that IC has better reification support, in that any constraint may be embedded in any other constraint expression, evaluating to that constraint's reified value.

- The primitives for accessing and manipulating the domains of variables are different; see section 3.2.7 on variable domain query predicates for details of IC's support for this.

3.1.3 Differences between IC and RIA

The main difference between IC's interval solving and RIA's is that IC is aware of and utilises the bounded real numeric type. This means bounded reals may appear in IC constraints, and IC variables may be unified with bounded reals (though direct unification is not recommended: it is preferable to use an equality constraint to do the assignment). In contrast, RIA will fail with a type error if bounded reals are used in either of these cases.

3.1.4 Notes about interval arithmetic

The main problem with using floating point arithmetic instead of real arithmetic for doing any kind of numerical computation or constraint solving is that it is only approximate. Finite precision means a floating point value may only approximate the intended real; it also means there may be rounding errors when doing any computation. Worse is that one does not know from looking at an answer how much error has crept into the computation; it may be that the result one obtains is very close to the true solution, or it may be that the errors have accumulated to the point where they are significant. This means it can be hard to know whether or not the answer one obtains is actually a solution (it may have been unintentionally

included due to errors), or worse, whether or not answers have been missed (unintentionally excluded due to errors).
Interval arithmetic is one way to manage the error problem. Essentially each real number is represented by a pair of floating point bounds. The true value of the number may not be known, but it is definitely known to lie between the two bounds. Any arithmetic operation to be performed is then done using these bounds, with the resulting interval widened to take into account any possible error in the operation, thus ensuring the resulting interval contains the true answer. This is the principle behind the bounded real arithmetic type.
Note that interval arithmetic does not guarantee small errors, it just provides a way of knowing how large the error may have become.
One drawback of the interval approach is that arithmetic comparisons can no longer always be answered with a simple "yes" or "no"; sometimes the only possible answer is "don't know". This is reflected in the behaviour of arithmetic comparators (=:=, >=, etc.) when applied to bounded reals which overlap each other. In such a case, one cannot know whether the true value of one is greater than, less than, or equal to the other, and so a delayed goal is left behind. This delayed goal indicates that the computation succeeded, contingent on whether the condition in the delayed goal is true. For example, if the delayed goal left behind was `0.2__0.4 >= 0.1__0.3`, this indicates that the computation should be considered a success only if the true value represented by the bounded real on the left is greater than or equal to that of the bounded real on the right. If the width of the intervals in any such delayed goals is non-trivial, then this indicates a problem with numerical accuracy. It is up to the user to decide how large an error is tolerable for any given application.

3.1.5 Interval arithmetic and IC

In order to ensure the soundness of the results it produces, the IC solver does almost all computation using interval arithmetic. As part of this, the first thing done to a constraint when it is given to the solver is to convert all non-integer numbers in it to bounded reals. Note that for this conversion, floating point numbers are assumed to refer to the closest representable float value, as per the type conversion predicate **breal/2**. This lack of widening when converting floating point numbers to bounded reals is fine if the floating point number is exactly the intended real number, but if there is any uncertainty, that uncertainty should be encoded by using a bounded real in the constraint instead of a float.
One of the drawbacks of this approach is that the user is not protected from the fundamental inaccuracies that may occur when trying to represent decimal numbers with floating point values in binary. The user should be aware therefore that some numbers given explicitly as part of their program may *not* be safely represented as a bounded real that spans the exact decimal value. e.g. `X $= 0.1` or equivalently `X is breal(0.1)`.
This may lead to unexpected results such as

```
[eclipse 2]: X $= 0.1, Y $= 0.09999999999999999, X $> Y.

X = 0.1__0.1
Y = 0.09999999999999992__0.09999999999999992
Yes (0.00s cpu)

[eclipse 3]: X $= 0.1, Y $= 0.099999999999999999, X $> Y.
```

```
No (0.00s cpu)
```

This potential source of confusion arises only with values which are explicitly given within a program. By replacing the assignment to Y with an expression which evaluates to the same real value we get

```
[eclipse 4]: X $= 0.1, Y $= 0.1 - 0.000000000000000001, X $> Y.

X = 0.1__0.1
Y = 0.09999999999999992__0.1


Delayed goals:
        ic : (0 > -1.3877787807814457e-17__-0.0)
Yes (0.00s cpu)
```

Note the delayed goal indicating the conditions under which the original goal should be considered to have succeeded.

3.1.6 Usage

To load the IC library into your program, simply add the following directive at an appropriate point in your code.

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

3.1.7 Arithmetic Expressions

The IC 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{0.36787944117144228 .. 1.0Inf}


Delayed goals:
...
Yes (0.01s cpu)
```

The IC library treats linear and non-linear constraints differently. Linear constraints are handled by a single propagator, whereas non-linear constraints are broken down into simpler ternary/binary/unary propagators.
Any relational constraint (`$=`, `$>=`, `#=`, etc.) can be reified simply by including it in an expression where it will evaluate to its reified truth value.
User-defined constraints may also be included in constraint expressions where they will be treated in a similar manner to user defined functions found in expressions handled by `is/2`. That is to say they will be called at run-time with an extra argument to collect the result.

Note, however, that user defined constraint/functions, when used in IC, should be deterministic. User defined constraints/functions which leave choice points may not behave as expected.
Variables appearing in arithmetic IC constraints at compile-time are assumed to be IC variables unless they are wrapped in an **eval/1** term. See section 3.1.7 for an more detailed explanation of usage.
The following arithmetic expression can be used inside the constraints:

`X` *Variables.* If `X` is not yet a interval variable, it is turned into one by implicitly constraining it to be a real variable.

`123` Integer constants. They are assumed to be exact and are used as is.

`0.1` Floating point constants. These are assumed to be exact and are converted to a zero width bounded reals.

`pi, e` Intervals enclosing the constants π and e respectively.

`inf` Floating point infinity.

`+Expr` Identity.

`-Expr` Sign change.

`+-Expr Expr` or `-Expr`. The result is an interval enclosing both. If however, either bound is infeasible then the result is the bound that is feasible. If neither bound is feasible, the goal fails.

`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. (Returns value between -pi/2 and pi/2.)

`rsqr(Expr)` Reverse of the sqr function. Equivalent to `+-sqrt(Expr)`.

`rpow(E1,E2)` Reverse of exponentiation. i.e. finds `X` in `E1 = X^E2`.

`sub(Expr)` A subinterval of Expr.

`sum(ExprList)` Sum of a list of expressions.

`min(ExprList)` Minimum of a list of expressions.

`max(ExprList)` Maximum of a list of expressions.

`and` Reified constraint conjunction. e.g. `B #= (X$>3 and X$<8)`

`or` Reified constraint disjunction. e.g. `B #= (X$>3 or X$<8)`

`=>` Reified constraint implication. e.g. `B #= (X$>3 => X$<8)`

`neg` Reified constraint negation. e.g. `B #= (neg X$>3)`

`$>`, `$>=`, `$=`, `$=<`, `$<`, `$\=`, `#>`, `#>=`, `#=`, `#=<`, `#<`, `#\=`, `>`, `>=`, `=:=`, `=<`, `<`, `=\=`, `and`, `or`, `=>`, `neg`
: Any arithmetic or logical constraint that can be issued as a goal may also appear within an expression.

  Within the expression context, the constraint evaluates to its reified truth value. If the constraint is entailed by the state of the constraint store then the (sub-)expression evaluates to `1`. If it is dis-entailed by the state of the constraint store then it evaluates to `0`. If its reified status is unknown then it evaluates to an integral variable `0..1`.

  Note: The simple cases (e.g. `Bool #= (X #> 5)`) are equivalent to directly calling the reified forms of the basic constraints (e.g. `#>(X, 5, Bool)`).

`foo(Arg1, Arg2 ... ArgN)`, `module:foo(Arg1, Arg2 ... ArgN)` Any terms found in the expression whose principle functor is not listed above will be replaced in the expression by a newly created auxiliary variable. This same variable will be appended to the term as an extra argument, and then the term will be called as `call(foo(Arg1, Arg2 ... ArgN, Aux))`. If no lookup module is specified, then the current module will be used.

This behaviour mimics that of **is/2**.

`eval(Expr)` See section 3.1.7 for an explanation of **eval/1** usage.

eval

The **eval/1** wrapper inside arithmetic constraints is used to indicate that a variable will be bound to an expression at run-time. This feature will only be used by programs which generate their constraints dynamically at run-time, for example.

```
broken_sum(Xs,Sum):-
    (
        foreach(X,Xs),
        fromto(Expr,S1,S2,0)
    do
        S1 = X + S2
    ),
    Sum $= Expr.
```

The above implementation of a summation constraint will not work as intended because the variable `Expr` will be treated like an IC variable when it is in fact the term `+(X1,+(X2,+(...)))` which is constructed in the for-loop. In order to get the desired functionality, one must wrap the variable `Expr` in an **eval/1**.

```
working_sum(Xs,Sum):-
    (
        foreach(X,Xs),
        fromto(Expr,S1,S2,0)
    do
        S1 = X + S2
    ),
    Sum $= eval(Expr).
```

3.2 Library Predicates

3.2.1 Domain constraints

Vars :: Domain Constrains Vars to take only integer or real values from the domain specified by Domain. Vars may be a variable, a list, or a submatrix (e.g. M[1..4, 3..6]); for a list or a submatrix, the domain is applied recursively so that one can apply a domain to, for instance, a list of lists of variables. Domain can be specified as a simple range Lo .. Hi, or as a list of subranges and/or individual elements (integer variables only). 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`.

::(Vars,Domain,Bool) Provides a reified form of the `::/2` domain assignment predicate. This reified `::/3` is defined only to work for one variable and only integer variables (unlike `::/2`), hence only the Domain formats suitable for integers may be supplied to this predicate.

For a single variable, V, the Bool will be instantiated to 0 if the current domain of V does not intersect with Domain. It will be instantiated to 1 iff the domain of V is wholly contained within Domain. Finally the Boolean will remain an integer variable in the range `0..1` if neither of the above two conditions hold.

Instantiating Bool to 1, will cause the constraint to behave exactly like `::/2`. Instantiating Bool to 0 will cause Domain to be excluded from the domain of all the variables in Vars where such an exclusion is representable. If such an integer domain is unrepresentable (e.g. `-1.0Inf .. -5, 5..1.0Inf`), then a delayed goal will be setup to exclude values when the bounds become sufficiently narrow.

Note that calling the reified form of `::` will result in the Variable becoming constrained to be integral, even if Bool is uninstantiated.

Further note that, like other reified predicates, `::` can be used infix in an IC expression e.g. `B #= (X :: [1..10])` is equivalent to `::(X, [1..10], B)`. See section 3.2.3 for more information of reified constraints.

Vars #:: Domain Constrains Vars to take only integer values from the domain specified by Domain. Vars may be a variable, a list, or a submatrix (e.g. M[1..4, 3..6]); for a list or a submatrix, the domain is applied recursively so that one can apply a domain to, for instance, a list of lists of variables. Domain can be specified as a simple range Lo .. Hi, or as a list of subranges and/or individual elements (integer variables only). Also allowed are the (untyped) symbolic bound values `inf`, `+inf` and `-inf`.

Vars $:: Domain Constrains Vars to take real values from the domain specified by Domain. Vars may be a variable, a list, or a submatrix (e.g. M[1..4, 3..6]); for a list or a submatrix, the domain is applied recursively so that one can apply a domain to, for instance, a list of lists of variables. Domain can only be specified as a simple range Lo .. Hi, in keeping with other implementations of this generic domain assignment predicate.

reals(Vars) Declares that the given variables are IC variables.

integers(Vars) Constrains the given variables to take integer values only.

3.2.2 Arithmetic constraints

Note that the integer forms of the constraints are essentially the same as the general forms, except that they check that all constants are integers and generally constrain all variables *and subexpressions* to be integral. Thus with integer constraints, the solver does very much behave like a traditional integer solver, with any temporary variables and intermediate results assumed to be integral. This means that it makes little sense to use many of the nonlinear functions available for use in expressions (e.g. sin, cos, ln, exp) in integer constraints. It also means that one should take care using such things as division: **X/2 + Y/2 #= 1** and **X + Y #= 2** are different constraints, with the former constraining X and Y to be even. That said, if all the constants and variables are integral already and the subexpressions clearly so as a consequence, then the integer (#) constraints and general ($) constraints may be used integerchangeably.

ExprX $= ExprY, ic:(ExprX =:= ExprY) ExprX is equal to ExprY. ExprX and ExprY are general expressions.

ExprX $>= ExprY, ic:(ExprX >= ExprY) ExprX is greater than or equal to ExprY. ExprX and ExprY are general expressions.

ExprX $=< ExprY, ic:(ExprX =< ExprY) ExprX is less than or equal to ExprY. ExprX and ExprY are general expressions.

ExprX $> ExprY, ic:(ExprX > ExprY) ExprX is strictly greater than ExprY. ExprX and ExprY are general expressions.

ExprX $< ExprY), ic:(ExprX < ExprY) ExprX is strictly less than ExprY. ExprX and ExprY are general expressions.

ExprX $\= ExprY, ic:(ExprX =\= ExprY) ExprX is not equal to ExprY. ExprX and ExprY are general expressions.

ExprX #= ExprY ExprX is equal to ExprY. ExprX and ExprY are constrained to be integer expressions.

ExprX #>= ExprY ExprX is greater than or equal to ExprY. ExprX and ExprY are constrained to be integer expressions.

ExprX #=< ExprY ExprX is less than or equal to ExprY. ExprX and ExprY are constrained to be integer expressions.

ExprX #> ExprY ExprX is greater than ExprY. ExprX and ExprY are constrained to be integer expressions.

ExprX #< ExprY ExprX is less than ExprY. ExprX and ExprY are constrained to be integer expressions.

ExprX #\= ExprY ExprX is not equal to ExprY. ExprX and ExprY are constrained to be integer expressions.

ac_eq(X, Y, C) Arc-consistent implementation of **X #= Y + C**. X and Y are constrained to be integer variables and to have "reasonable" bounds. C must be an integer.

The comparison constraints `=:=/2`, `>=/2`, `=</2` and `=\=/2` have the same syntax as the standard ECL^{i}PS^e built-in comparison operators (and those of other constraint solvers). Unless explicitly qualified, the ECL^{i}PS^e built-ins are used. To use these constraints without the need to qualify them, use the alternative dollar-syntax.

3.2.3 Reified constraints

As mentioned earlier, when constraints appear in an expression context, then they evaluate to their reified truth value. Practically this means that the constraints are posted in a passive *check but do not propagate* mode, whereby no variable domains are modified but checks are made to see if the constraint has become entailed (necessarily true) or dis-entailed (necessarily false).
The simplest and arguably most natural way to reify a constraint is to place it in an expression context (i.e. on either side of a `$=`, `$>=`, `#=`, etc.) and assign its truth value to a variable. For example:

```
TruthValue #= (X $> 4).
```

It is also possible to use the 3 argument form of the constraint predicates where the third argument is the reified truth value, for example:

```
$>(X, 4, TruthValue).
```

But in general the previous form is recommended as it can be easily extended to handle the truth values of a combination of constraints, by using the infix operators `and` (logical conjunction), `or` (logical disjunction) and `=>` (logical implication) or the prefix operator `neg` (logical negation). e.g.:

```
TruthValue #= (X $> 4 and Y $< 6).
```

Again, as mentioned earlier, there are a number of reified connectives which can be used to combine reified constraints using logical operations on their truth values.

and/2 Reified constraint conjunction. e.g. `B #= (X $> 3 and X $< 8)` or `X $> 3 and X $< 8`

or/2 Reified constraint disjunction. e.g. `B #= (X $> 3 or X $< 8)` or `X $> 3 or X $< 8`

=>/2 Reified constraint implication. e.g. `B #= (X $> 3 => X $< 8)` or `X $> 3 => X $< 8`

neg/1 Reified constraint negation. e.g. `B #= (neg X $> 3)` or `neg X $> 3`

Enforcing constraints

The logical truth value of a constraint, when reified, can be used to enforce the constraint (or its negation) during search.
The following three examples are equivalent:

```
X $> 4.
```

```
B #= (X $> 4), B=1.
```

```
B #= (X $=< 4), B=0.
```

By unifying the value of the reified truth value, the constraint changes from being *passive* to being *active*. Once actively true (or actively false) the constraint will prune domains as though it had been posted as a simple non-reified constraint.

User-defined reified constraints

Reified constraints are implemented using the the 3 argument form of the constraint predicate if it exists (and it does exist for the arithmetic relation constraints).
User-defined constraints will be treated as reifiable if they appear in an expression context and as such should provide forms where the last argument is the reified truth value reflected into a variable.
The user-defined constraint should behave as follows depending on the state of the reified variable.

Reified variable is unbound When the reified variable is unbound, the constraint should not perform any domain reduction on its arguments, but should check to see if the constraint has become entailed or dis-entailed, setting the reified variable to `1` or `0` respectively.

Reified variable is bound to 0 In the event that the reified variable becomes bound to `0` then the constraint should actively propagate its negation.

Reified variable is bound to 1 In the event that the reified variable becomes bound to `1` then the constraint should actively propagate its normal semantics.

3.2.4 Miscellaneous constraints

alldifferent(Vars) Constrains all elements of a list to be different from all other elements of the list.

element(Index, List, Value) Constrains Value to be the Index'th element of the list of integers List.

3.2.5 Integer labeling predicates

These predicates can be used to enumerate solutions to a set of constraints over integer variables. For optimisation, see also the **branch_and_bound** library.

indomain(Var) Instantiates an integer IC variable to an element of its domain.

labeling(Vars) Instantiates all IC variables in a list to elements of their domains.

search(Vars, Arg, Select, Choice, Method, Options) Instantiates the variables Vars by performing a search based on the parameters provided by the user.

3.2.6 Real domain refinement predicates

These predicates can be used to locate real solutions to a set of constraints. They are essentially the same as those that were available in RIA; more details of the algorithms used can be found in section 3.2.10.

locate(Vars, Precision) Locate solution intervals for Vars by splitting and search. Precision indicates how accurate the intervals have to be (in absolute or relative terms) before splitting stops.

locate(Vars, Precision, LinLog) As per locate/2, but LinLog specifies wither linear (`lin`) or logarithmic (`log`) splitting should be used. (**locate/2** is equivalent to calling **locate/3** with `log` as the third argument.)

locate(LocateVars, SquashVars, Precision, LinLog) As per **locate/3**, but also applies the squashing algorithm to SquashVars both before splitting commences, and then again after each split.

squash(Vars, Precision, LinLog) Refine the intervals of Vars by the squashing algorithm.

3.2.7 Variable query predicates

These predicates allow one to retrieve various properties of an IC variable (and usually work on ground numbers as well).

is_solver_var(Var) Succeeds if an only if Var is an IC variable.

is_solver_type(Term) Succeeds if an only if Term is an IC variable or a number.

get_solver_type(Var, Type) Returns whether Var is an integer variable or a real variable.

get_bounds(Var, Lo, Hi) Returns the current bounds of Var.

get_min(Var, Lo) Returns the current lower bound of Var.

get_max(Var, Hi) Returns the current upper bound of Var.

get_float_bounds(Var, Lo, Hi) Returns the current bounds of Var as floats.

get_integer_bounds(Var, Lo, Hi) Returns the current bounds of the integer variable Var (infinite bounds are returned as floats). Constrains Var to be integral if it isn't already.

get_finite_integer_bounds(Var, Lo, Hi) Returns the current (finite) bounds of the integer variable Var. Constrains Var to be finite and integral if it isn't already.

get_domain_size(Var, Size) Returns the number of elements in the IC domain for Var. Currently Var needs to be of type integer.

get_domain(Var, Domain) Returns a ground representation of the current IC domain for Var.

get_domain_as_list(Var, Domain) Returns a list of all the elements in the IC domain for Var. Currently Var needs to be of type integer.

get_median(Var, Median) Returns the median of the interval of Var.

get_delta(Var, Delta) Returns the width of the interval of Var.

is_in_domain(Var, Value) Succeeds if and only if Value is contained in the current domain of Var.

is_in_domain(Var, Value, Result) Binds Result to 'yes', 'no' or 'maybe' depending on whether Value is in the current domain of Var.

delayed_goals_number(Var, Number) Returns the number of delayed goals suspended on the IC attribute. This approximates the number of IC constraints that Var is involved in.

3.2.8 Propagation threshold predicates

With interval constraint propagation, it is sometimes useful to limit propagation for efficiency reasons. In IC, this is controlled by the propagation threshold. The way it works is that for non-integer variables, bounds are only changed if the absolute and relative changes of the bound exceed this threshold (i.e. small changes are suppressed). This means that constraints over real variables are only guaranteed to be consistent up to the current threshold (over and above any normal widening which occurs).
Note that 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.
The default threshold is 1e-8.

get_threshold(Threshold) Returns the current propagation threshold.

set_threshold(Threshold) Sets the propagation threshold. Note that if the threshold is reduced using this predicate (requiring a higher level of precision), the current state of the system may not be consistent with respect to the new threshold. If it is important that the new level of precision be realised for all or part of the system before computation proceeds, **set_threshold/2** should be used instead.

set_threshold(Threshold, WakeVars) Sets the propagation threshold, with re-computation. If the threshold has been reduced, all constraints suspended on the bounds of the variables in the list WakeVars are woken.

3.2.9 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{2.1925824014821353 .. 2.1925824127108307}

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{2.1925824014821353 .. 2.1925824127108307}

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 .. 7.0}
```

```
Delayed goals:
    ...
yes.
```

3.2.10 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 (in either absolute or relative terms). 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 3.2.10) 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 3.2.10) is applied once 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.

Squash algorithm

A stronger propagation algorithm is also included. This is built upon the normal bound consistency. It guarantees that, if you take any variable and restrict its range to a small domain near one of its bounds, the original bound 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).

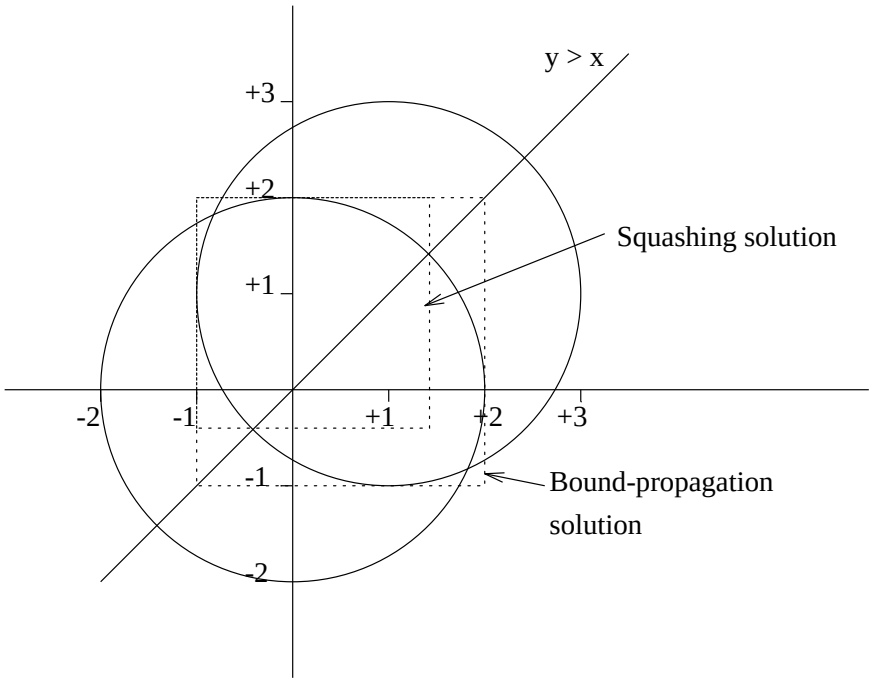

Figure 3.1: Propagation with Squash algorithm (example)

```
[eclipse 2]: 4 $>= X^2 + Y^2, 4 $>= (X-1)^2+(Y-1)^2, Y $>= X.

Y = Y{-1.0000000000000004 .. 2.0000000000000004}
X = X{-1.0000000000000004 .. 2.0000000000000004}

Delayed goals:
    ...
yes.
```

The bound-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 2]: 4 $>= X^2 + Y^2, 4 $>= (X-1)^2+(Y-1)^2, Y $>= X,
                squash([X,Y], 1e-5, lin).

X = X{-1.0000000000000004 .. 1.4142135999632601}
Y = Y{-0.41421359996326 .. 2.0000000000000004}

Delayed goals:
    ...
yes.
```

3.2.11 Obtaining Solver Statistics

(Using the facilities described in this section requires importing the **ic_kernel** module. Also, since they depend on the internals of the IC library, the details presented here are subject to change without notice.)
Often it is difficult to know where the solver spends its time. The library has built-in counters which keep track of the number of times various events occur:

ic_lin_create The number of linear constraints set up.

ic_lin_prop The number of times a linear constraint is propagated.

ic_uni_prop/ic_bin_prop/ic_tern_prop The number of times a non-linear (unary/binary/ternary) operator is propagated.

ic_split The number of domain splits in locate/2,3,4.

ic_squash The number of squash attempts in squash/3 or locate/4.

Users who wish to track activity within their own programs may (if they wish) use the same mechanism. New event types can be registered (see below) and actions recorded by calling the **ic_event(Event)** predicate.
The counters are controlled using the primitives:

ic_stat(on)

ic_stat(off) Enables/disable collection of statistics. Default is off.

ic_stat(reset) Reset statistics counters.

ic_stat(print) Print statistics counters to the standard output stream.

ic_stat_get(-Stat) Returns a list of CounterName=CounterValue pairs, summarising the computation since the last reset.

ic_event(+Name) Records the fact that the named event has happened.

ic_stat_register_event(+Name,+Description) Registers a new event type and sets the counter to 0. This allows user-defined predicates to record their own events within the same framework.

3.3 General Guidelines for the Use of the IC library

Whilst IC has been designed to provide a flexible, consistent and yet powerful framework for many sorts of constraint satisfaction problems, it can not be all things to all people.
There are circumstances under which IC will not propagate all possible information, for reasons of efficiency.
It is the purpose of this section to point out ways that may help IC to solve problems more efficiently.
Typical constraint satisfaction problems are solved by iteratively propagating information from basic constraints until no more propagation can take place (i.e. a fixed point has been reached), and then reducing the domain of a variable so as to prompt more propagation.

As with most constraint solvers the propagation provided by the builtin constraints is rarely able to solve large problems completely without the need for some form of search. A number of factors affect the speed of the propagation phase.

1. The size of the initial domains. Smaller domains typically result in propagation reaching a fixed point sooner. So give explicit initial domains to as many variables as possible.

2. Integer domains allow more propagation. An important point to note here is that (as in mathematics) IC treats integers as a strict subset of the reals, and as such the integer domain `0 .. 100` contains significantly fewer values than the real domain `0.0 .. 100.0`. With this in mind, IC attempts to infer integrality where possible (e.g. the sum of two integer variables is constrained to be integer), however integer domains (where applicable) should be used in user code.

   The difference becomes apparent when dealing with strict inequalities, for example.

```
[eclipse 4]: reals([X]), X $> 5.

X = X{5.0 .. 1.0Inf}


Delayed goals:
        ic : (X{5.0 .. 1.0Inf} > 5)
Yes (0.00s cpu)
```

   Note that the lower bound of X is still five despite the fact that X has been constrained to be strictly greater than five. Further note the presence of a delayed goal which will fail should X be constrained to be exactly five.

```
[eclipse 5]: integers([X]), X $> 5.

X = X{6 .. 1.0Inf}
Yes (0.00s cpu)
```

   In this example since X is known to be integral, the lower bound of X can be set to 6, as there are no integers between five and six.

3.4 User defined constraints

The library **ic_kernel** provides a number of facilities useful for implementing IC constraints or otherwise extending the facilities provided by the standard IC library.
While the **ic_kernel** library exposes the structure of the IC attribute to the programmer (see below), accessing it directly is strongly discouraged (if for no other reason, the internals of IC may continue to evolve). For accessing information about a variable and its domain, use the predicates described earlier in section 3.2.7 “Variable query predicates”. For modifying a variable, it is particularly important to go through the access predicates, in order to make sure that the internal state remains consistent, appropriate constraints are scheduled for execution as a result of the change, etc. The predicates available for modifying a variable are discussed in the next section.

3.4.1 Modifying variable domains

When using IC variables in normal code, one would typically use the `$\=`, `$=<` and `$>=` family of constraints to (resp.) remove a value, reduce the upper bound or increase the lower bound of a variable.
While these constraints are good for normal CSP solving, they have a number of properties which may be less desirable when writing new constraints. In particular, they may leave unwanted delayed goals behind and may perform extra propagation before returning (it may be desirable to perform all required bound updates before allowing further propagation to occur).
To give the constraint writer more control over such matters, special predicates exist in the **ic_kernel** module which allow direct modification of the domain without the waking of goals (they are scheduled for execution but not actually executed). These predicates generally accept an IC variable, a non-IC variable (which will be constrained to make it a real IC variable) or a number.
Full details on these predicates can be found in the reference manual; they are listed here for completeness. Note that with the exception of **impose_bounds/3** none of the goals call **wake/0**, so the programmer is free to do so at a convenient time.

impose_min/2 Set the lowerbound.

impose_max/2 Set the upperbound.

impose_bounds/3 Sets both upper and lower bounds.

exclude/2 Excludes an integer from an integral variable.

exclude_range/3 Excludes a range of integers from an integral variable.

set_var_type/2 Makes the variable be of the given type.

set_vars_type/2 Like set_var_type, but works for lists and submatrices of variables as well.

3.4.2 The IC attribute

The IC attribute is a meta-term which is attached to all variables which take part in IC constraints. **ic_kernel** defines the IC attribute as a structure of the following form:

```
ic with [var_type:Type,
         lo:Lo,
         hi:Hi,
         bitmap:Bitmap,
         min:SuspMin,
         max:SuspMax,
         hole:SuspHole,
         type:SuspType
        ]
```

This structure holds

var_type The type of the variable. This defaults to 'real' but may become 'integer' after an explicit call to **integers/1**, by being included in an integer constraint (e.g. #=) or by inferences made during constraint propagation.

lo The lower bound of the variable's domain, as a float.

hi The lower bound of the variable's domain, as a float.

bitmap Where relevant, a bitmap represent the integer domain; where not relevant it holds the atom `undefined`.

min Suspension list of goals to be woken on lower bound changes.

max Suspension list of goals to be woken on upper bound changes.

hole Suspension list of goals to be woken when a value is removed from the middle of a domain. Such removals only happen for integer variables whose domain is finite.

type Suspension list of goals to be woken when a variable's type becomes more constrained, i.e. when a variable goes from being real to being integer.

The suspension list names can be used in **suspend/3** and related predicates to denote an appropriate waking condition.
The attribute of a domain variable can be accessed with the predicate **get_ic_attr/2**.
As noted above, direct access and manipulation of the attribute is discouraged; use the access predicates instead.

Chapter 4

Additional Finite Domain Constraints

4.1 Various Constraints on Lists

The library **ic_global** implements a number of constraints over lists of integer variables. It is loaded using one of

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

The following predicates are provided

alldifferent(+List)
A version of alldifferent/1 with strong propagation.

alldifferent(+List, ++Capacity)
Like alldifferent/1, but every value may occur Capacity times.

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.

lexico_le(+List1, +List2)
Imposes a lexicographic ordering between the two lists.

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

ordered_sum(++List, +Sum)
The list elements are ordered and their sum is Sum.

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.

sorted(?List, ?Sorted)
Sorted is a sorted permutation of List.

sorted(?List, ?Sorted, ?Positions)
Sorted is a sorted permutation of List and Positions is a list whose elements indicating the position of each unsorted list element within the sorted list.

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

4.2 Cumulative Constraint and Resource Profiles

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

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

cumulative(+StartTimes, +Durations, +Resources, ++ResourceLimit)
A cumulative scheduling constraint. StartTimes, Durations and Resources are lists of equal length N of integer variables 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 integer variables 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.

4.3 Edge-finder

The libraries **ic_edge_finder** and **ic_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(ic_edge_finder)).
```

to get a weaker variant with quadratic complexity, or

```
:- use_module(library(ic_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 integer variables 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 integer variables 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 resource usage are not known in advance. The edge-finder algorithm can make use of this information to derive bound updates.

Chapter 5

The Integer Sets Library

5.1 Introduction

The *fd_sets* library is a solver for constraints over the domain of finite sets of integers. Unlike *conjunto*, it cannot deal with sets elements that are not integers. On the other hand, fd_sets is usually faster for integer sets than conjunto.

5.2 Ground Integer Sets

(Ground) integer sets are simply sorted, duplicate-free lists of integers e.g.

```
SetOfThree = [1,3,7]
EmptySet = []
```

Lists which contain non-integers, are unsorted or contain duplicates, are not sets in the sense of this library.

5.3 Set Variables

5.3.1 Declaring

Set variables are variables which can eventually take a ground integer set as their value. They are characterized by a lower bound (the set of elements that are definitely in the set) and an upper bound (the set of elements that may be in the set). A set variable can be declared as follows:

```
SetVar :: []..[1,2,3,4,5,6,7]
```

If the lower bound is the empty set (like in this case) this can be written as

```
SetVar subset [1,2,3,4,5,6,7]
```

If the lower bound is the empty set and the upper bound is a set of consecutive integers, one can also declare it like

```
intset(SetVar, 1, 7)
```

which is equivalent to the above.
The predicates to declare sets are:

?Set :: ++Lwb..++Upb Set is an integer set within the given bounds

intset(?Set, +Min, +Max) Set is a set containing numbers between Min and Max

intsets(?Sets, ?N, +Min, +Max) Sets is a list of N sets containing numbers between Min and Max

5.3.2 Printing

Set variables are by default printed in a particular way, e.g.

```
?- X :: [2,3]..[1,2,3,4], write(X).
X{[2, 3] \/ ([] .. [1, 4]) : _308{[2 .. 4]}}
```

The curly brackets contain

1. the lower bound of the set
2. the union symbol
3. the set of optional values (that may or may not be in the set)
4. a colon
5. a finite domain indicating the admissible cardinality for the set

5.3.3 Domain Access

potential_members(?Set, -List) List is the list of elements of whose membership in Set is currently uncertain

set_range(?Set, -Lwb, -Upb) Lwb and Upb are the current lower and upper bounds on Set

5.4 Constraints

5.4.1 Membership

?X in ?Set The integer X is member of the integer set Set

?X notin ?Set The integer X is not a member of the integer set Set

membership_booleans(?Set, ?BoolArr) BoolArr is an array of booleans describing Set

5.4.2 Cardinality

#(?Set, ?Card) Card is the cardinality of the integer set Set

5.4.3 Set Relations

difference(?Set1, ?Set2, ?Set3) Set3 is the difference of the integer sets Set1 and Set2

?Set1 disjoint ?Set2 The integer sets Set1 and Set2 are disjoint

?Set1 includes ?Set2 Set1 includes (is a superset) of the integer set Set2

intersection(?Set1, ?Set2, ?Set3) Set3 is the intersection of the integer sets Set1 and Set2

?Set1 sameset ?Set2 The sets Set1 and Set2 are equal

?Set1 subset ?Set2 Set1 is a subset of the integer set Set2

symdiff(?Set1, ?Set2, ?Set3) Set3 is the symmetric difference of the integer sets Set1 and Set2

union(?Set1, ?Set2, ?Set3) Set3 is the union of the integer sets Set1 and Set2

5.4.4 N-ary Set Relations

all_disjoint(+Sets) Sets is a list of integers sets which are all disjoint

all_union(+Sets, ?SetUnion) SetUnion is the union of all the sets in the list Sets

all_intersection(+Sets, ?SetIntersection) SetIntersection is the intersection of all the sets in the list Sets

5.4.5 Set Weights

weight(?Set, ++ElementWeights, ?Weight) According to the array of element weights, the weight of set Set1 is Weight

5.5 Set Expressions

In most positions where a set or set variable is expected one can also use a set expression. A set expression is composed from ground sets (integer lists), set variables, and the following set operators:

```
Set1 /\ Set2       % intersection
Set1 \/ Set2       % union
Set1 \ Set2        % difference
```

When such set expressions occur, they are translated into auxiliary **intersection/3**, **union/3** and **difference/3** constraints, respectively.

5.6 Search Support

The **insetdomain/4** predicate can be used to enumerate all ground instantiations of a set variable, much like **indomain/1** in the finite-domain case.

insetdomain(?Set, ?CardSel, ?ElemSel, ?Order) Instantiate Set to a possible value

5.7 Example

The following program computes so-called Steiner triplets. These are triplets of numbers from 1 to N such that any two triplets have at most one element in common.

```
:- lib(fd_sets).
:- lib(fd).

steiner(N, Sets) :-
        NB is N * (N-1) // 6,           % compute number of triplets
        intsets(Sets, NB, 1, N),        % initialise the set variables
        ( foreach(S,Sets) do
            #(S,3)                      % constrain their cardinality
        ),
        ( fromto(Sets,[S1|Ss],Ss,[]) do
            ( foreach(S2,Ss), param(S1) do
                #(S1 /\ S2, C),         % constrain the cardinality
                C #<= 1                 % of pairwise intersections
            )
        ),
        label_sets(Sets).               % search

label_sets([]).
label_sets([S|Ss]) :-
        insetdomain(S,_,_,_),
        label_sets(Ss).
```

Here is an example of running this program

```
?- steiner(9,X).

X = [[1, 2, 3], [1, 4, 5], [1, 6, 7], [1, 8, 9],
     [2, 4, 6], [2, 5, 8], [2, 7, 9], [3, 4, 9],
     [3, 5, 7], [3, 6, 8], [4, 7, 8], [5, 6, 9]] More? (;)
```

Chapter 6

The Symbolic Domain Library

6.1 Introduction

The *ic_symbolic* library is a solver for constraints over ordered symbolic domains. It is implemented on top of library(ic) (see 3), by mapping symbolic domains to finite integer domains. There are also several mixed-domain constraints, which have both symbolic and integer arguments.

6.2 Domains and Domain Variables

This library uses the **domain** feature provided by the ECLiPSe kernel. This means that domains need to be declared. The declaration specifies the domain values and their order. For example:

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

declares a domain with name 'weekday' and values 'mo', 'tu' etc. The domain values are implicitly ordered, with 'mo' corresponding to 1, until 'su' corresponding to 7. Domain values must be unique within one ECLiPSe module, i.e. a symbolic value can belong to at most one domain.
A variable of a declared domain can then be created using

```
?- X &:: weekday.
X = X{[mo, tu, we, th, fr, sa, su]}
Yes (0.00s cpu)
```

or multiple variables using

```
?- [X,Y,Z] &:: weekday.
X = X{[mo, tu, we, th, fr, sa, su]}
Y = Y{[mo, tu, we, th, fr, sa, su]}
Z = Z{[mo, tu, we, th, fr, sa, su]}
Yes (0.00s cpu)
```

6.3 Basic Constraints

The following constraints implement the basic relationships between two domain values. The constraints require their arguments to come from identical domains, otherwise an error is raised.

?X &= ?Y X is the same as Y

?X &\= ?Y X is different from Y

?X &> ?Y X is strictly before Y in the domain order

?X &< ?Y X is strictly after Y in the domain order

?X &=< ?Y X is the same as Y, or before Y in the domain order

?X &>= ?Y X is the same as Y, or after Y in the domain order

shift(?X,?C,?Y) Y is C places above X in the domain order. X and Y have symbolic domains, C has an integer domain.

rotate(?X,?C,?Y) like shift/3 but wraps at domain boundary.

element(?Index,++List,?Value) Value occurs List at position Index. Value has a symbolic domain, Index has an integer domain. List is a number of symbolic domain values.

For example

```
?- [X, Y] &:: weekday, X &< Y.
X = X{[mo, tu, we, th, fr, sa]}
Y = Y{[tu, we, th, fr, sa, su]}
Yes (0.00s cpu)

?- X &:: weekday, X &=< we.
X = X{[mo, tu, we]}
Yes (0.00s cpu)
```

6.4 Global Constraints

A number of global constraints are available which directly correspond (and are in fact implemented via) their counterparts in lib(ic_global):

alldifferent(+List) All list elements are different

occurrences(++Value,+List,?N) Value occurs N times in List

atmost(++N,+List,++Value) Value occurs at most N times in List

6.5 Internals

Internally, symbolic domains are mapped to integer ranges from 1 up to the number of domain elements. The first value in the domain declaration corresponds to 1, the second to 2 and so on. Similarly, symbolic domain variables can be mapped to a corresponding IC integer variable. This mapping is accessible through the predicate **symbol_domain_index/3**:

```
?- symbol_domain_index(fr, D, I).
D = weekday
I = 5
Yes (0.00s cpu)

?- X &:: weekday, symbol_domain_index(X, D, I).
X = X{[mo, tu, we, th, fr, sa, su]}
D = weekday
I = I{1 .. 7}
Yes (0.00s cpu)

?- X &:: weekday, X &\= we, symbol_domain_index(X, D, I).
X = X{[mo, tu, th, fr, sa, su]}
D = weekday
I = I{[1, 2, 4 .. 7]}
Yes (0.00s cpu)
```

The integer variable I mirrors the domain of the symbolic variable X and vice versa.

6.6 Extending and Interfacing this Library

Because of the mapping of symbols to integers, new constraints over symbolic variables can be implemented simply by placing numeric (IC) constraints on the corresponding integer variables.
Similarly, the facilities of the ic_search library can be exploited when working with symbolic variables. Instead of labeling the symbolic variables, one can use the various facilities of ic_search to label the corresponding integer variables instead.

Chapter 7

Propia - A Library Supporting Generalised Propagation

7.1 Overview

Propia is the name for the implementation of Generalised Propagation in ECL^{i}PS^e.
Generalised propagation is *not* restricted to integer 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 [15, 14, 16]. This section focusses on how to use it. Further examples of the use of Propia are distributed with ECL^{i}PS^{e}in the `doc/examples/propia/` directory . A simple demonstration of Propia in action on Lewis Carroll's Zebra problem can be run by compiling `zebra.pl` and issuing the query `lib(ic), zebra(Houses,ic)` . An slightly more complex application of Propia to crossword generation can be run by compiling `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.

7.2 Invoking and Using Propia

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

```
?- lib(propia).
```

A goal, such as `member(X,[1,2,3])`, 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 7.3 below. In this section we shall use `Goal infers most`, which infers as much information as possible, given the loaded constraint solvers. If the `IC` solver is loaded, then `IC` information is extracted, and Propia reduces the domains to achieve arc-consistency.
We first show the behaviour of the original goal:

```
?- member(X, [1, 2, 3]).
X = 1
Yes (0.00s cpu, solution 1, maybe more)
X = 2
Yes (0.02s cpu, solution 2, maybe more)
X = 3
Yes (0.02s cpu, solution 3)
```

Constraint propagation is invoked by `infers most`:

```
?- lib(ic).
...
?- member(X, [1, 2, 3]) infers most.
X = X{1 .. 3}
Yes (0.00s cpu)
```

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:

```
?- 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 #<, so after compiling this simple program:

```
notin3to6(X) :- X#<3.
notin3to6(X) :- X#>6.
```

the predicate can be used as a constraint:

```
?- X :: 1 .. 10, notin3to6(X) infers most.
X = X{[1, 2, 7 .. 10]}
Yes (0.00s cpu)
```

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:

```
noclash(ST1,D1,ST2,D2) :- ST1 #>= ST2+D2.
noclash(ST1,D1,ST2,D2) :- ST2 #>= ST1+D1.
```

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

```
?- lib(ic).
...

?- [ST1, ST2] :: 1 .. 10, noclash(ST1, 5, ST2, 7) infers most.
ST1 = ST1{[1 .. 5, 8 .. 10]}
ST2 = ST2{[1 .. 3, 6 .. 10]}
There is 1 delayed goal.
Yes (0.00s cpu)
```

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:

```
land(true,true,true).
land(true,false,false).
land(false,true,false).
land(false,false,false).
```

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

```
?- land(X, Y, Z) infers most, X = Y.
Z = X
X = X
Y = X
There is 1 delayed goal.
Yes (0.00s cpu)
```

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.

```
/*** product(Name,#Component1,#Component2,Profit). **/
product(1,1,19,1).
product(2,2,17,2).
product(3,3,15,3).
product(4,4,13,4).
product(5,10,8,5).
product(6,16,4,4).
product(7,17,3,3).
product(8,18,2,2).
product(9,19,1,1).
```

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=[1,5,1]`), to the number of each component required to build all the products in the list and the profit (for `[1,5,1]`, `Comp1=12` and `Comp2=46` and `Profit=7`).

```
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).
```

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:

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

Now `sum` refuses to make any choices:

```
?- sum([V1, V2, V3], Comp1, Comp2, Profit).
V1 = V1{1 .. 9}
```

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

```
V2 = V2{1 .. 9}
V3 = V3{1 .. 9}
Comp1 = Comp1{3 .. 57}
Comp2 = Comp2{3 .. 57}
Profit = Profit{3 .. 15}
There are 9 delayed goals.
Yes (0.01s cpu)
```

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`:

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

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.

```
?- solve(P, 9, 95, 95, 40).
P = [1, 4, 5, 5, 5, 5, 5, 5, 5]
Yes (0.03s cpu, solution 1, maybe more)
```

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 `IC` library is loaded, and the constraint has only one remaining variable.

7.3 Approximate Generalised Propagation

The syntax *Goal infers most* can also be varied to invoke different levels of Generalised Propagation. Other alternatives are *Goal infers ic*, *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 `IC` integer domains.

```
p(1,a).
p(2,f(Z)).
p(3,3).
```

```
?- p(X, Y) infers most.
X = X{1 .. 3}
Y = Y
There is 1 delayed goal.
Yes (0.00s cpu)

?- X :: 1 .. 3, p(X, Y) infers most.
X = X{1 .. 3}
Y = Y
There is 1 delayed goal.
Yes (0.00s cpu)

?- p(2, Y) infers most.
Y = f(_326)
There is 1 delayed goal.
Yes (0.00s cpu)
```

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.

```
?- p(X, Y) infers unique.
X = X
Y = Y
There is 1 delayed goal.
Yes (0.00s cpu)
```

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

```
?- p(X,X) infers unique.
X = 3
Yes (0.00s cpu)
```

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

```
?- p(2, X) infers unique.
X = f(Z)
Yes (0.00s cpu)
```

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.

```
?- p(1, X) infers consistent.
X = X
There is 1 delayed goal.
Yes (0.00s cpu)
```

```
?- p(1, a) infers consistent.
Yes (0.00s cpu)

?- p(1, X) infers consistent, X = b.
No (0.00s cpu)
```

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), *ic* and *ic_symbolic*. The `IC` library is loaded by `lib(ic)` and the symbolic library by `lib(ic_symbolic)`. These libraries are described elsewhere. If both libraries are loaded, then `infers most` extracts information from unification, `IC` domains and symbolic domains. For example:

```
p(f(X),1) :- X *>=0, X *=< 10.
p(f(X),1) :- X=12.
```

with the above program

```
?- p(X, Y) infers most.
X = f(_338{0.0 .. 12.0})
Y = Y{[1, 2]}
There is 1 delayed goal.
Yes (0.00s cpu)
```

The approximation `infers ic` is 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 ic`:

```
?- p(X, Y) infers ic.
X = f(_353{0.0 .. 12.0})
Y = Y{[1, 2]}
There is 1 delayed goal.
Yes (0.00s cpu)
```

One rather special approximation langue is `infers ac`, where `ac` stands for arc-consistency. This has similar semantics to `infers ic`, but is implemented very efficiently using the built-in `element` constraint of the `IC` 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,7,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 7.

Chapter 8

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 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 8.9.

8.1 Introduction

Constraint handling rules (CHRs, CHR home page http://www.pst.informatik.uni-muenchen.de/-fruehwir/chr-intro.html) [6] 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.

8.2 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).`

8.3 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) [8] (`*kl-one*`),
- temporal reasoning (over time points and intervals) [7] (`*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.
```

8.4 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.

8.4.1 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 8.8), 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.

8.4.2 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.
```

8.5 More on the CHR Language

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

8.5.1 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).
```

8.5.2 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).

8.5.3 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 8.8).

8.5.4 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".

8.6 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.
```

8.7 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.

8.7.1 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.

8.7.2 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`.

8.8 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.

8.8.1 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.

8.9 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.
- The CHR are run at the same priority before and after suspensions. Priorities can be specified for CHR constraints.
- 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.

8.9.1 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.

8.9.2 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.
...
```

8.9.3 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.

8.9.4 Semantics

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.
```

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).

Execution Priority

The priority at which an ECH rule is executed depends on the 'active' constraint, i.e. the constraint that triggered the execution of the rules. Normally, the ECH rules are executed at the *default* priority, but a different priority can be associated with a constraint when it is declared, specifying the priority at which the ECH rules will be executed when that constraint is the active constraint.

```
:- constraints chr_labeling/0:at_lower(1).
```

this specifies that if `chr_labeling/0` was the active constraint, then the rules will be executed at a lower priority than the default. The priorities are mapped to the priority system of ECL^{i}PS^e, and `at_lower(1)` maps to a priority one lower than the default, so that ECH rules executing at the default priority will execute first. This is particularly useful for labelling, as this allow the other ECH constraints to be executed during the labelling process rather than afterwards.
The priority specified can be `at_lower(N)`, `at_higher(N)`, or `at_absolute_priority(N)`. For `at_lower(N)`, the priority is the default + N; for `at_higher(N)`, it is the default - N. `at_absolute_priority(N)` sets the priority to N, regardless of the default, and its use is not recommended. The available priorities are from 1 (highest) to 11 (lowest). The default priority is initially set to 9, but can be changed with the `chr_priority` option. Note that the priority at which the rules will run at is determined at compile time, and changing the default priority will only affect new constraints compiled after the change. It should therefore only be used in a directive before any of the ECH rules.
This behaviour is different from the old `chr` library, and from older versions of `ech` library, where the rules ran at different priorities before and after suspension. This can lead to different

behaviours with the same rule, either with other constraints solvers, or even with other CHR rules, as a woken CHR executes at much higher priority than the initial run. With the current `ech` execution, the rules are executed at the same priority before and after suspension, for the same active constraint. The default priority is set at 9 so that it is very likely to be lower than the priority used in other constraint solvers. The user is now allowed to alter the priority of specific ECH constraints to allow the user more control so that for example a labelling rule can run at a lower priority than the other constraints.

8.9.5 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`.
Another new option, `default_chr_priority`, allows the default priority to be changed, e.g.

```
:- option(default_chr_priority, 6).
```

changes the default priority to 6, so the compiler would generate new CHR code which defaults to this priority (unless overridden in the constraints declaration). The available values are from 1 to 11.
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,

8.9.6 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 9

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

9.1 Usage

This library allows the use of an external mathematical programming (LP, MIP or quadratic) solver from within ECL^{i}PS^e. It provides a largely solver-independent API to the programmer, so many programs will run with any supported external solver.
See section 9.10 for more details on the supported solvers.
The most generic way to load the library is:

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

This will try to load an appropriate external solver available on the computer.
It is also possible to request a specific solver explicitly, see section 9.10 for details.
Note that the ECL^{i}PS^e library described here is just an interface to an external solver. In order to be able to use it, you need to have access to a solver supported by the library. For commercial solvers, this requires a licence for the solver on your machine. For more details, see section 9.10.

9.2 Eplex Instances

In this chapter, the problem passed to the external solver will be referred to as an *eplex problem*. An eplex problem consists of a set of linear arithmetic constraints, whose variables have bounds and may possibly have integrality constraints. The external solver will solve such a problem by optimising these constraints with respect to an objective function.
With the eplex library, it is possible to have more than one eplex problem within one program. The simplest way to write such programs with the library is through *Eplex Instances*. An eplex instance is an instance of the eplex solver, to which an eplex problem can be sent. An external *solver state* can be associated with each eplex instance, which can be invoked to solve the eplex problem. Declaratively, an eplex instance can be seen as a compound constraint consisting of all the variables, their bounds, and constraints of the eplex problem.
Like other solvers, each eplex instance has its own module. To use an eplex instance, it must first be declared, so that the module can be created. This is done by:

eplex_instance(+Name)

This predicate will initialise an eplex instance `Name`. Once initialised, a `Name` module will exist, to which the user can post the constraints for the eplex problem and setup and use the external solver state to solve the eplex problem. Normally, this predicate should be issued as a directive in the user's program, so that the program code can refer to the instance directly in their code. For example:

```
:- eplex_instance(instance).
```

For convenience, the eplex library declares `eplex` as an eplex instance when the library is loaded.

9.2.1 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 is transferred to the external solver state if the state has been setup. If it has not, the constraint delays and is transferred to the external solver state when it is setup. 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

As with all predicates defined for an eplex instance, these constraints should be module-qualified with the name of the eplex instance. In the following they are shown qualified with the `eplex` instance. Other instances can be used if they have been declared using **eplex_instance/1**.

EplexInstance: (X $= Y)

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

EplexInstance: (X $>= Y)

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

EplexInstance: (X $=< Y)

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

9.2.2 Linear Expressions

The following arithmetic expression can be used inside the constraints:

X Variables. If X is not yet a problem variable, it is turned into one via an implicit declaration `X $:: -1.0Inf..1.0Inf`.

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.

9.2.3 Bounds

Bounds for variables can be given to an eplex instance via the `$::/2` constraint:

EplexIntance: Vars $:: Lo..Hi Restrict the external solver to assign solution values for the eplex problem within the bounds specified by Lo..Hi. Passes to the external solver the bounds for the variables in Vars. Lo, Hi are the lower and upper bounds, respectively. Note that the bounds are only passed to the external solver if they would narrow the current bounds, and failure will occur if the resulting interval is empty. Note also that the external solver does not do any bound propagation and will thus not change the bounds on its own. The default bounds for variables are notionally -1.0Inf..1.0Inf (where infinity is actually defined as the solver's notion of infinity).

9.2.4 Integrality

The difference between using an LP vs. an MIP solver is made by declaring integrality to the solver via the integers/1 constraint:

EplexInstance:integers(Vars) Inform the external solver to treat the variables Vars as integral. It does not impose the integer type on Vars. However, when a typed_solution is retrieved (via lp_get/3 or lp_var_get/3), this will be rounded to the nearest integer.

Note that unless eplex:integers/1 (or lp_add/3, see section 9.4.2) is invoked, any invocation of the eplex external solver (via lp_solve/2, lp_probe/3 or lp_demon_setup/5) will only solve a continuous relaxation, even when problem variables have been declared as integers in other solvers (e.g. ic).

Note that all the above constraints are local to the eplex instance; they do not place any restrictions on the variables for other eplex instances or solvers. Failure will occur only when inconsistency is detected within the same eplex instance, unless the user explicitly try to merge the constraints from different solvers/eplex instance.

9.2.5 Solving Simple Eplex Problems

In order to solve an eplex problem, the eplex instance must be set up for an external solver state. The solver state can then be invoked to solve the problem. The simplest way to do this is to use:

EplexInstance:eplex_solver_setup(+Objective) This predicate creates a new external solver state and associates it with the eplex instance. Any arithmetic, integrality and bound constraints posted for this eplex instance are collected to create the external solver state. After this, the solver state can be invoked to solve the eplex problem.

Objective is either min(Expr) or max(Expr) where Expr is a linear expression (or quadratic, if supported by the external solver).

EplexInstance:eplex_solve(-Cost) Explicitly invokes the external solver state. Any new constraints posted are taken into account. If the external solver can find an optimal solution to the eplex problem, then the predicate succeeds and Cost is instantiated to the optimal value. If the problem is infeasible (has no solution) or unbounded (Cost is not bounded by the constraints), then the predicate fails.

9.2.6 Examples

Here is a simple linear program, handled by the predefined eplex instance 'eplex':

```
:- lib(eplex).

lp_example(Cost) :-
     eplex: eplex_solver_setup(min(X)),
     eplex: (X+Y $>= 3),
     eplex: (X-Y $= 0),
     eplex: eplex_solve(Cost).
```

The same example using a user-defined eplex instance:

```
:- lib(eplex).
:- eplex_instance(my_instance).

lp_example(Cost) :-
     my_instance: eplex_solver_setup(min(X)),
     my_instance: (X+Y $>= 3),
     my_instance: (X-Y $= 0),
     my_instance: eplex_solve(Cost).
```

Running the program gives the optimal value for Cost:

```
[eclipse 2]: lp_example(Cost).

Cost = 1.5
```

Note that if the `eplex` eplex instance is used instead of `my_instance`, then the `eplex_instance/1` declaration is not necessary.
By declaring one variable as integer, we obtain a Mixed Integer Problem:

```
:- lib(eplex).
:- eplex_instance(my_instance).

mip_example(Cost) :-
     my_instance: (X+Y $>= 3),
     my_instance: (X-Y $= 0),
     my_instance: integers([X]),
     my_instance: eplex_solver_setup(min(X)),
     my_instance: eplex_solve(Cost).

....
[eclipse 2]: mip_example(Cost).

Cost = 2.0
```

The cost is now higher because X is constrained to be an integer. Note also that in this example, we posted the constraints before setting up the external solver, whereas in the previous example we set up the solver first. The solver set up and constraint posting can be done in any order. If `integers/1` constraints are only posted after problem setup, the problem will be automatically converted from an LP to a MIP problem.
This section has introduced the most basic ways to use the eplex library. We will discuss more advanced methods of using the eplex instances in section 9.3.

9.3 Advanced Use of Eplex Instances

9.3.1 Obtaining Solver State Information

The black-box interface binds both the objective value (Cost) and the problem variables by bindings these variables. On the other hand, **eplex_solve/1** binds the objective value, but does not bind the problem variables. These values can be obtained by:

EplexInstance:eplex_var_get(+Var, +What, -Value) Retrieve information about the solver state associated with the eplex instance for the variable `Var`. If `What` is `solution` or `typed_solution`, then the value assigned to this variable by the solver state to obtain the optimal solution is returned in `Value`. `solution` returns the value as a float, and `typed_solution` returns the value as either a float or a rounded integer, depending on if the variable was constrained to an integer in the eplex problem.

EplexInstance:eplex_get(+What, -Value) Retrieve information about solver state associated with the eplex instance. This returns information such as the problem type, the constraints for the eplex problem. See the reference manual for more details.

EplexInstance:eplex_set(+What, +Value) Set a solver option for the eplex instance.

EplexInstance:eplex_write(+Format, +File) Write out the problem in the the eplex instance's solver state to the file File in format Format. The writing is done by the external solver. Use the use_var_name(yes) option in **eplex_solver_setup/4** so that the written file uses ECL^{i}PS^{e}variable names. Also the `write_before_solve` option of

eplex_solver_setup/4 can be used to write out a problem just before it is solved by the external solver: this allows problem to be written in places where eplex_write/2 cannot be added (e.g. for probing problems)..

EplexInstance:eplex_read(+Format, +File) Read a MP problem in the file File in format Format into a solver state, and associate the solver with the eplex instance. No solver must already be setup for the eplex instance. The solver state that is setup can only be triggered explicitly.

So for the simple MIP example:

```
:- lib(eplex).
:- eplex_instance(my_instance).

mip_example2([X,Y], Cost) :-
     my_instance: (X+Y $>= 3),
     my_instance: (X-Y $= 0),
     my_instance: integers([X]),
     my_instance: eplex_solver_setup(min(X)),
     my_instance: eplex_solve(Cost),
     my_instance: eplex_var_get(X, typed_solution, X),
     my_instance: eplex_var_get(Y, typed_solution, Y).

....
[eclipse 2]: mip_example2([X,Y],C).

X = 2
Y = 2.0
C = 2.0
```

In the example, only X is returned as an integer, as Y was not explicitly constrained to be an integer.
Note that if there are multiple eplex instances, and a variable is shared between the instances, then the solver state for each instance can have a different optimal value to the variable.

9.3.2 Creating Eplex Instances Dynamically

So far, we have shown the use of `eplex_instance/1` as a directive to declare an eplex instance. For some applications, it might be necessary to create eplex instances dynamically at run-time. The can be done by calling `eplex_instance/1` at run-time. In this case, the instance name should *not* be used to module-qualify any predicates in the code, since this will raise a compiler warning complaining about an unknown module.

```
new_pool(X,Y) :-  % INCORRECT
   eplex_instance(pool),
   pool: (X $>= Y), % will generate a warning
   ...
```

Of course, in the above code, the instance name `pool` is already known at compile time, so it can always be declared by a directive.

If the name is truly generated dynamically, this can be done as follows:

```
new_pool(Pool,X,Y) :-
    eplex_instance(Pool),
    Pool: (X $>= Y),
    ....
```

9.3.3 Interface for CLP-Integration: Solver Demons

To implement hybrid algorithms where a run of a simplex/MIP solver is only a part of the global solving process, the black-box model presented above is not appropriate anymore. With eplex instances, we can call `eplex_solve/1` repeatedly to re-solve the problem, perhaps after adding more constraints to the problem or after changes in the variable bounds. However, the solver must be invoked explicitly. We require more sophisticated methods of invoking the solver. This can be done by setting up a solver demon, and specifying the conditions in which the demon is to wake up and invoke the external solver.

EplexInstance:eplex_solver_setup(+Objective, -Cost, +ListOfOptions, +TriggerModes)

This is a more sophisticated set up for a new solver state than `eplex_solver_setup/1` (in fact eplex_solver_setup/1 is a special case of eplex_solver_setup/4). The main idea is that a list of trigger conditions are specified in `TriggerModes`, and along with setting up the solver state, a demon goal is created which is woken up when one of the specified trigger condition is met. This demon goal will then invoke the solver, with any constraints posted to the eplex instance since the solver was last invoked taken into account, to re-solve the problem.

The `ListOfOptions` is a list of solver options for setting up the solver state. Some of these affect the way the external solver solves the problem, such as if presolve should be applied before solving the problem. See the reference manual for **eplex_solver_setup/4** for details on the available options and trigger modes.

As the solver is designed to be invoked repeatedly, it is inappropriate to directly bind `Cost` to the objective value. Instead, the objective value is exported as a bound to Cost: 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 variable bounds or added constraints. Note that the bound update is done only if the solution is optimal. Note also that Cost is not automatically made a problem variable, and thus may not have bounds associated with in. In order for the bounds information not to be lost, some bounds should be given to Cost (e.g. making it a problem variable (but this might introduce unnecessarily self-waking on bounds change), or via another solver with bounds (e.g. ic)).

Note that when a solver demon runs frequently on relatively small problems, it can be important for efficiency to switch the external solver's presolving off for this demon as part of the `ListOfOptions` during the setup of the problem to reduce overheads.

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 (we use the eplex instance in this example):

```
simplex :-
    eplex:eplex_solver_setup(min(0), C, [solution(no)], [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 a finite domains solver, e.g.

```
[eclipse 2]: eplex:(X+Y+Z =< 1),
    eplex_solver_setup(min(0), C, [solution(no)], [bounds]),
    K = 2.

No (0.00s cpu)
```

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. To do this, we need to give the objective 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 5]: ic: (Cost $:: -1.0Inf..1.0Inf),
      eplex:(X+Y $=< 1), eplex:(Y+Z $=< 1), eplex:(X+Z $=< 1),
      eplex:eplex_solver_setup(max(X+Y+Z), Cost, [solution(no)], [bounds]).

X = X{-1e+20 .. 1e+20}
Y = Y{-1e+20 .. 1e+20}
Z = Z{-1e+20 .. 1e+20}
Cost = Cost{-1.0Inf .. 1.500001}


Delayed goals:
        lp_demon(prob(...), ...)
Yes (0.00s cpu)
```

(Note that the ranges for X, Y and Z is -1e+20 .. 1e+20 as 1e+20 is this external solver's notion of infinity).
If the variable bounds change subsequently, the solver will be re-triggered, improving the cost bound to 1.3:

```
[eclipse 6]: ic: (Cost $:: -1.0Inf..1.0Inf),
      eplex:(X+Y $=< 1), eplex:(Y+Z $=< 1), eplex:(X+Z $=< 1),
      eplex:eplex_solver_setup(max(X+Y+Z), Cost, [solution(no)], [bounds]),
      eplex:(Y =< 0.3).

X = X{-1e+20 .. 1e+20}
Z = Z{-1e+20 .. 1e+20}
Cost = Cost{-1.0Inf .. 1.300001}
Y = Y{-1e+20 .. 0.3}


Delayed goals:
        lp_demon(prob(...), ...)
Yes (0.00s cpu)
```

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

9.3.4 Probing Using a Different Objective

The external mathematical programming solvers often provides the facility for the user to change the problem being solved. This includes the addition or removal of constraints, and the changing of the objective function. We have already seen how extra constraints can be added. As ECL^{i}PS^e is a logic programming language, removal of constraints is automatically achieved by backtracking. We do not allow the user to explicitly remove constraints that have been collected by the external solver, as this makes the problem non-monotonic. For the same reason, we do not allow the objective function to be changed.[1] However, we do allow the problem (including the objective function) to be *temporarily* changed in certain specified ways. This allows the problem to be 'probed' with these changes:

EplexInstance:eplex_probe(+Probes, -Cost)

Similar to eplex_solve/1, but the problem is first temporarily modified as specified in Probes before the optimisation. The Cost value is instantiated to the objective value for this new modified problem, and any solution state requested are also updated.

9.3.5 Destroying the Solver State

EplexInstance:eplex_cleanup

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. The solver is disassociated with the eplex instance, and any outstanding constraints not yet collected by the solver are removed, with a warning to the user. In effect, the eplex instance is reinitialised.

[1] However, some monotonic changes are allowed in the low-level interface, for implementing column generation, see section 9.4.5.

Note that this is a non-logical operation. Backtracking into code before `eplex_cleanup/0` will not restore the solver state, and any attempt to reuse the solver state will not be possible (the execution will abort with an error). Normally, it is recommended to let ECL^{i}PS^e perform the cleanup automatically, for instance when execution fails across the solver setup, or when an unused solver state handle gets garbage collected. However, calling eplex_cleanup/0 may cause resources (memory and licence) to be freed earlier.

9.3.6 Eplex Instance Interface Example: definition of optimize/2:

A black-box setup-and-solve predicate **optimize/2** can be defined as:

```
optimize(OptExpr, ObjVal) :-
        eplex:eplex_solver_setup(OptExpr),
        eplex:eplex_solve(ObjVal),
        eplex:eplex_get(vars, VArr),
        eplex:eplex_get(typed_solution, SolutionVector),
        VArr = SolutionVector,                  % do the bindings
        eplex:eplex_cleanup.
```

A solver state is set up for the eplex instance `eplex`, to allow constraints that were previously posted to `eplex` to be collected. This happens once the solver is invoked by `eplex_solve/1`. If there is a solution, the solution vector is obtained, and the variables are instantiated to those solutions.

9.4 Low-Level Solver Interface

For many applications, the facilities presented so far should be appropriate for using Simplex/MIP through ECL^{i}PS^e. However, sometimes it may be more convenient or efficient to directly access the solver state instead of going through the abstraction of the eplex instances. This section describes lower level operations like how to set up solvers manually. In fact, these lower level predicates are used to implement the predicates provided with eplex instances.
These predicates accesses the external solver state via a handle, which is returned when the solver state is set up, and subsequently used to access a particular solver state by the other predicates. The handle should be treated as a opaque data structure that is used by the eplex library to refer to a particular solver state.

9.4.1 Setting Up a Solver State

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

This is used to set up a demon solver, and `eplex_solver_setup/4` calls this predicate. There is one extra argument compared to `eplex_solver_setup/4`: the solver state handle `Handle`, which is returned by this predicate when the new solver state is created. The other arguments are the same as in `eplex_solver_setup/4`, except that there is an additional option in `ListOfOptions`: `collect_from/1`. This is used to specify which, if any, eplex instance the solver state should be collecting constraints from. If an eplex instance is specified (as `pool(Instance)`), then the solver state is associated with that instance. If the eplex instance is *not* to be associated with an eplex instance, `none` should be given as the argument to `collect_from`. This allows a solver state to be set up without the overhead of an eplex instance.

The solver state will not collect any constraints automatically when it is invoked; instead the constraints must be added explicitly via the handle (using `lp_add_constraints/3`).
By default, the external solver is invoked once after set up by `lp_demon_setup`, if any `TriggerModes` is specified. Otherwise, the solver is not invoked and the predicate returns after set up.

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

This is an even lower-level primitive, setting up a solver state without any automatic triggering. It creates 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 in other solvers (e.g. using **ic: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 (or quadratic) expression. ListOfOptions is a list of solver options, the same as for **lp_demon_setup/5** and **eplex_solver_setup/4**, except for the `collect_from` and `initial_solve` options, which are specific for the demon solvers.

9.4.2 Adding Constraints to a Solver State

Constraints can be added directly to a solver state without posting them to an eplex instance. This is done by:

lp_add_constraints(+Handle, +Constraints, +NewIntegers)

Add new constraints (with possibly new variables) to the solver state represented by Handle The new constraints will be taken into account the next time the solver is run. The constraints will be removed on backtracking.
The constraints are first normalised, and simple constraints filtered out (as discussed in section 9.2.1) before they are added to the external solver (by calling lp_add/3 described below).

lp_add(+Handle, +NewNormConstraints, +NewIntegers)

This adds the constraints (both linear and intergrality) to the external solver represented by Handle. The linear arithmetic constraints must be normalised. Note that it is possible to add trivial constraints, which would be filtered out by the higher level `lp_add_constraints/3` using this predicate. Integrality constraints on non-problem variables are filtered out and a warning given.

lp_add_vars(+Handle, +Vars)

This adds the variables in Vars to the external solver state represented by Handle. The variables should not contain variables which are already problem variables. The variables are given the default bounds of -infinity..infinity.

lp_var_set_bounds(+Handle, +Var, ++Lo,++Hi)

This updates the bounds for the problem variable Var in the external solver state represented by Handle. Failure occurs if Var is not a problem variable.

9.4.3 Running a Solver State Explicitly

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 and lp_var_get/4.
The set of constraints considered by the solver is the one given when the solver was created plus any new constraints that were added (e.g by lp_add_constraints/3) in the meantime.
If there was an error condition, or limits were exceeded, lp_solve/2 raises an event. See section 9.8 for details.

lp_probe(+Handle, +Probes, -Cost)

Similar to lp_solve/2, but optimize for a modified problem as specified by Probes. This is the predicate called by **eplex_probe/2**

9.4.4 Accessing the Solver State

In section 9.3.1, we discussed how solver state information can be accessed via the eplex instance. Here are the lower level predicates that directly access this information via the solver state's handle:

lp_get(+Handle, +What, -Value)

Retrieve information about solver state and results. See the reference manual description of `lp_get/3` for a detailed description of the available values for `What`.
For example, it is possible to obtain the solution values from the last successful invocation of the external solver using the following:

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

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

Retrieve information about solver state represented by Handle, related to a specific variable Var. Again, see the reference manual for the available parameters.

lp_var_get_bounds(+Handle, +Var, -Lo, -Hi)

Retrieve the bounds of the problem variable Var from the solver state represented by Handle.

reduced_cost_pruning(+Handle,?GlobalCost)

This predicate implements a technique to prune variable bounds based on a global cost bound and the reduced costs of some solution to a problem relaxation. The assumptions are that there is a global problem whose cost variable is GlobalCost, and that Handle refers to a linear relaxation of this global problem. The pruning potentially affects all variables involved in the relaxed problem.

9.4.5 Expandable Problem and Constraints

We provide low-level primitives to 'expand' an eplex problem. Such a problem is considered to have as yet unspecified components in the objective function and posted constraints. These constraints are known as expandable constraints. The as yet unspecified component involve variables that have not yet been added to the problem. When these variables are added, coefficients for the variables can be added to the expandable constraints, as well as the objective function. These primitives are the basis for implementing **column generation**, and are used by the column generation library, lib(colgen).
These primitives modify an existing eplex problem *non-monotonically*, and can only be used on problems that are not represented by an eplex instance, and was not setup as a demon solver (i.e. no trigger conditions are specified).

lp_add_constraints(+Handle, +Constraints, +Ints, -Idxs)

This adds expandable constraints Constraints to the solver state represented by Handle. The predicate returns a list of indicies for these constraints in Idxs. The indicies are used to refer to the constraints when new variables are added to expand the problem.

lp_add_columns(+Handle, +Columns)

This expands the problem by adding new variables (columns) to the solver state represented by Handle. Columns is a list of variable:column-specification pair, where variable is the variable to be added as a new column, and column-specification the specification for the non-zero components of the column, i.e. coefficients for the expandable constraints (referred to using the index obtained from lp_add_constraints/4) and the objective for this variable.

9.4.6 Changing Solver State Settings

In addition to accessing information from the solver state, some options (a subset of those specified during solver set up) can be changed by:

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. See the reference manual for details.

9.4.7 Destroying a Solver State

lp_cleanup(+Handle)

Destroy the specified solver state, free all memory, etc. If the solver state is associated with an eplex handle, the solver state is disassociated with the eplex instance. However, unlike **eplex_cleanup/0**, the outstanding constraints not yet collected by the solver is not removed. As with `eplex_cleanup/0`, care should be taken before using this non-logical predicate.

9.4.8 Miscellaneous Predicates

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.

lp_write(+Handle, +Format, +File)

Write out the problem in the solver state represented by Handle to the file File in format Format.

normalise_cstrs(+Constraints, -NormConstraints, -NonlinConstr)

where Constraints is a list of terms of the form X `$=` Y, X `$>=` Y or X `$=<` Y 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.

9.5 Multiple Solver States

This library allows multiple solver states to be maintained in the same program. Each solver state represents an eplex problem. For the external solver, each solver state is completely independent. For ECL^{i}PS^e, the solver states may share variables in the constraints or objective functions. The eplex library maintains separate solution values for each solver state, and it is up to the user to reconcile these solution values if they are different.
When two eplex variables are unified, then the library ensures that the now single variable maintains the eplex values from both variables. The one exception is when two variables from the same solver state is unified. In this case, an equality constraint between the two variables is sent to the solver state, but the user can only obtain one eplex value from the unified variable, even though in the external solver, the variable is still represented as two variables (columns in the matrix).
It is possible to turn off this automatic sending of the equality constraints by specifying 'no' for the option `post_equality_when_unified` (in solver setup, or via `eplex_set/2`). The reason is that some solvers automatically perform unification when they know that two variables are the same. For example, for the constraint `X $= Y + Z`, if `Y` becomes 0, then `X` and `Z` may be unified by the solver maintaining the constraint. If the same constraint was also posted to the eplex solver state, then there is no need to send the redundant constraint. However, if the external solver state did not have the constraint, then it can become inconsistent with that of ECL^{i}PS^e if the equality constraint is not sent. Therefore, only turn off sending of equality constraints if you are certain you know what you are doing.

9.6 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` is directed to ECL^{i}PS^e's `error` stream, `warning_channel` to `warning_output` stream, while `result_channel` and `log_channel` are suppressed. To see the output on these channels, do for instance

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

9.7 Dealing with Large and Other Non-standard Numbers

In many external solvers, infinities or very large numbers are not handled directly. Instead, these solvers define a large (floating point) number to be infinity. However, the problem that is sent to the external solver may contain values greater than the solver's notion of infinity. This is handled in the following way:

- If a variable's range extends beyond the solver's infinity, the range is rounded down.
- If some coefficient (constant) in the problem is outside the solver's range, an out of range error would be raised when this is detected (and the problem is not passed to the external solver).

In addition, ECL^{i}PS^e supports numeric types that are not generally available, e.g. bounded real and rational. These are converted into floating point numbers before they are passed to the external solver.

9.8 Error Handling

The external solver's optimization can abort without completely solving the problem, because of some error, or some resource limit was reached. Eplex classifies these into the following cases, with default ways of handling them:

suboptimal This means that a solution was found but it may be suboptimal. The default behaviour is to print a warning and succeed.

unbounded This means that the problem is unbounded. The default behaviour is to bind Cost to infinity (positive or negative depending on the optimisation direction), print a warning and succeed. CAUTION: No solution values are computed when the problem is unbounded, so unless the problem was set up with the solution(no) option, an error will occur when trying to continue as if the optimisation had succeeded.

unknown This means that due to the solution method chosen, it is unknown whether the problem is unbounded or infeasible. The default behaviour is to print a warning and fail (even though this may be logically wrong!).

abort Some other error condition occurred during optimisation. The default behaviour is to print an error and abort.

The default behaviours can be overridden for each problem by giving a user defined goal to handle each case during problem setup in eplex_solver_setup/4 (lp_setup/4, lp_demon_setup/5, or later with eplex_set/2 or lp_set/3) as an option. If given, the user defined goal will be executed instead. The user defined handler could for instance change parameter settings and call lp_solve again.
The default behaviour is implemented by raising the events `eplex_suboptimal`, `eplex_unbounded`, `eplex_unknown` and `eplex_abort` for the different abort cases. These events can themselves be redefined to change the default behaviours. However, as this changes the behaviour globally, it is not recommended.

9.9 Solver Behaviour Differences

In general, an MP problem can have more than one optimal solution (i.e. different sets of assignments to the problem variables that gives the optimal objective value). Any of these solutions is correct, and the external solver will return one of them. It is possible for a different solver (or even a different version of the same solver) to return another of these solutions. If the user's program uses the solution values, then it is possible that the detailed behaviour of the program could depend on the solver being used.
The solution that is returned can also depend on the detailed settings of the floating point unit of the processor. Thus changing some of these settings may change the solution that is returned. It is thus possible for eplex to give different solutions on the same machine and solver if these settings are changed (e.g. when ECL^{i}PS^e is embedded into a Java application).

9.10 Solver Specific Information

The external solvers currently supported by the eplex library are:

- XPRESS-MP, a product of Dash Optimization (www.dashoptimization.com)
- CPLEX, a product of ILOG (www.ilog.com)

Both companies offer academic licences at discounted prices, or academic partner programs.
To load a specific solver explicitly, use:

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

The first line explicitly requests the CPLEX solver, the second line explicitly requests the XPRESS-MP solver. Note that these solvers must be available for your machine for the above to work.

9.10.1 Versions and Licences

All the solvers supported by the library come in various versions. The set of supported solver versions may vary between different releases of ECL^{i}PS^e; please refer to the release notes. Depending on which solver licence you have, 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 the appropriate 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. The file contains lines of the form

```
licence(Hostname, Solver, Version, LicStr, LicNum).
```

For example, if you have CPLEX version 7.5 on machine `workhorse`, and XPRESS-MP version 15.20 (with the license file located in `/my/xpress/license`) on machine `mule`, and your Internet domain is `+icparc.ic.ac.uk`, you would add the lines

```
licence('workhorse.icparc.ic.ac.uk', cplex, '75', ■, 0).
licence('mule.icparc.ic.ac.uk', xpress, '1520', '/my/xpress/license', 0).
```

The hostname must match the result of get_flag(hostname,H), converted to an atom (this is normally the complete Internet domain name, rather than just the machine). Version is formed from the concatenation of the major and minor version numbers. The meaning of LicStr and LicNum depends on the optimizer: For CPLEX with normal licenses, they are unused (the environment variable `ILOG_LICENSE_FILE` should be set to the CPLEX license file `access.ilm` as usual). For XPRESS-MP, LicStr is a string specifying the directory where licence file is located (overrides value of XPRESS environment variable). LicNum is unused. If a machine has more than one licence and lib(eplex) is called, the first one listed in `eplex_lic_info.ecl` will be used.

9.10.2 Access to External Solver's Control Parameters

The external solver has a number of control parameters that affect the way it works. These can be queried and modified using the **lp_get/2**, **eplex_get/2**, **lp_get/3**, and **lp_set/2**, **eplex_set/2**, **lp_set/3** predicates respectively:

lp_get(+Handle, optimizer_param(+ParamName), -Value)

Retrieve the value of a control parameter for the external solver for the problem represented by Handle. These paramters are solver specific; see **lp_get/3** for more details..

EplexInstance:eplex_get(optimizer_param(+ParamName), -Value)

Like lp_get/3, but get a control parameter for the external solver associated with the specified eplex instance.

lp_get(optimizer_param(+ParamName), -Value)

Retrieve the global value of a control parameter for the external solver. The paramters and the exact meaning of 'global' is solver specific: if the solver does not have global parameters, this gets the global default value, rather than the globally applicable value. The parameters are as in lp_get/3.

lp_set(+Handle, optimizer_param(+ParamName), +Value)

Set a control parameter for the external solver for the problem represented by Handle. If the external solver does not have problem specific parameters, this will raise an unimplemented functionality exception. The parameters are as in lp_get/3.

EplexInstance:eplex_set(optimizer_param(+ParamName), +Value)

Like lp_set/3, but set a control parameter for the external solver associated with the specified eplex instance.

lp_set(optimizer_param(+ParamName), +Value)

Set a control parameter for the external solver for the problem globally. If the external solver does not have global parameters, this will set the global default for the parameter. The parameters are as in lp_get/3.

lp_get(optimizer, -Value) and lp_get(optimizer_version, -Value)

Retrieve the name (currently 'cplex' or 'xpress') and version of the external optimizer. This can be used to write portable code even when using solver-specific settings:

```
( lp_get(optimizer, xpress) ->
    ( lp_get(optimizer_version, Version), Version >= 13 ->
        lp_set(Handler, optimize_param(maxnode), 100)
    ;
        lp_set(Handler, optimize_param(maxnod), 100)
    )
; lp_get(optimizer, cplex) ->
    lp_set(Handler, optimize_param(node_limit), 100)
), ...
```

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 [5].

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', 'ic', 'fd_sets' 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.

?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.

?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]: fd:(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]: fd:(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]: fd:(X::1..5), X tent_set 7, X tent_set 3.
X = X{fd:[1..5], repair:3}
```

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.

10.2.6 Copying

If a variable with a repair attribute is copied using **copy_term/2** or similar, the repair attribute is stripped. If you wish the copy to have the same tentative value as the original, you will need to call **tent_get/2** and **tent_set/2** yourself.

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:

Constraint r_conflict ConflictSet

This is the simplest form of annotation. **r_conflict/2** 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:

```
fd:(Capacity >= sum(Weights))  r_conflict  cap_cstr
```

Note that using different conflict sets for different groups of constraints will often make the search algorithm easier and more efficient. A second allowed form of the **r_conflict/2** 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.

Constraint r_conflict_prop ConflictSet

In addition to what **r_conflict/2** does, the **r_conflict_prop/2** 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.

conflict_vars(-Vars)

When a variable becomes untenable, it appears in the set of conflict variable, when it becomes tenable, it 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.

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.

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:

-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.

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.

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]:
        fd:([X,Y,Z]::1..3),                     % the problem variables
        fd:(Y #\= X) r_conflict_prop confset,   % state the constraints
        fd:(Y #\= Z) r_conflict_prop confset,
        fd:(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/1 predicate in the 'repairfd' library file.

```
repair(ConflictSet) :-
    ( conflict_vars([C|_]) ->       % label conflict
        indomain(C),                %  variables first
        repair(ConflictSet)
    ; conflict_constraints(ConflictSet, [C|_]) ->
        term_variables(C, Vars),    % choose one variable in
        deleteffc(Var,Vars, _),     %  the conflict constraint
        Var tent_get Val,
        (Var = Val ; fd:(Var #\= Val)),
        repair(ConflictSet)
    ;                               % 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.

Index

Bibliography

[1] F. Ajili and H. El Sakkout. LP probing for piecewise linear optimization in scheduling. Programme and papers presented at CPAIOR'01: `www.icparc.ic.ac.uk/cpAIOR01/`, 2001.

[2] N. Beldiceanu and E. Contjean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 12:97–123, 1994. `citeseer.nj.nec.com/beldiceanu94introducing.html`.

[3] A. Bockmayr and T. Kasper. Branch and infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998.

[4] H. H. El Sakkout and M. G. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4):359–388, 2000.

[5] Hani El Sakkout. *Improving Backtrack Search: Three Case Studies of Localized Dynamic Hybridization*. PhD thesis, Imperial College, London, June 1999.

[6] 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.

[7] T. Fruehwirth. Temporal reasoning with constraint handling rules. Technical Report Core-93-8, ECRC Munich, Germany, January 1993.

[8] 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.

[9] T. Frühwirth. Theory and practice of constraint handling rules. *Logic Programming*, 37(1-3):95–138, 1988.

[10] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[11] C. Holzbauer. OFAI clp(q,r) Manual. Technical Report TR-95-09, Austrian Research Institute for AI, Vienna, 1995.

[12] ILOG. CPLEX. `www.ilog.com/products/cplex/`, 2001.

[13] C. Le Pape and P. Baptiste. Resource constraints for preemptive job-shop scheduling. *Constraints*, 3(4):263–287, 1998.

[14] 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.

[15] T. Le Provost and M.G. 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.

[16] T. Le Provost and M.G. Wallace. Generalized constraint propagation over the CLP Scheme. *Journal of Logic Programming*, 16(3-4):319–359, July 1993. Special Issue on Constraint Logic Programming.

[17] Olivier Lhomme, Arnaud Gotlieb, Michel Rueher, and Patrick Taillibert. Boosting the interval narrowing algorithm. In *Joint International Conference and Symposium on Logic Programming*, pages 378–392, 1996.

[18] L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. *Lecture Notes in Computer Science*, 1330, 1997.

[19] Dash Optimization. XPRESS-MP. `www.dash.co.uk/`, 2001.

[20] P. Van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 1995.

[21] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[22] M.G. Wallace and J. Schimpf. Finding the right hybrid algorithm - a combinatorial meta-problem. *Annals of Mathematics and Artificial Intelligence*, 34(4):259–270, 2002.