CS-Prolog II

The ML Linear Solver

Version 0.1

Supplement to User's Manual

Contents

1. Introduction

The Constraint Logic Programming paradigm had been introduced by J.Jaffar et al. [J.Jaffar, S.Michaylov, *Methodology and Implementation of a CLP System*, in J.L.Lassez (ed.) Logic Programming — Proceedings of the 4th International Conference — Vol.1, MIT Press, Cambridge, MA, 1987].

An early version of CS-Prolog (around 1990) had included support for CLP over real numbers using a solver based on linear programming methods. With the evolution of CS-Prolog towards multiple processes, this support was discontinued.

Recently the need of providing a general mechanism for attaching CLP solvers to CSP-II had arisen. During the implementation of this feature, we tried to revitalize the old solver as a testing tool. Due to internal reasons (lack of time, lost documentation, unavailability of the original developers, etc.), this revitalization has been only a partial success; there are some serious deficiencies and limitations. Nevertheless, we decided to distribute the tool 'as is'; we shall refer to it as the **ML solver** throughout the discussion. It can be used as an example for those developing their own solver, or just for experimenting with the CLP paradigm. We hope that in the future we'll have the opportunity to improve the solver so that it can be used for real work.

This document describes the particular details of the implementation of the interface predicates for the ML solver, not covered by the general description in the CSP-II User's Manual.

The ML solver supports all normal CLP interface predicates, and no other interface predicates are defined for it. (This must not be surprising given that the general interface has been defined as a generalization of the original interface of the ML solver.)

The central concept of the CLP paradigm is *constraint*. A constraint is a Prolog term, normally containing one or more variables, passed to the solver (in a **clp_constraint/1** call). Each unbound variable seen by the solver is associated with a newly created *problem variable* maintained by the solver. The association itself is represented by binding the variable to a term of type **constrained variable** (which is added to the type system as part of the CLP extension).

The solver incrementally builds the current (satisfiable) set of constraints from the individual constraints passed to it. Each constraint describes some condition that must be satisfied, in a solver-specific form. The solver will accept a new constraint only if adding it to the current set will result in a satisfiable set again. In addition, if unification involving a constrained variable is attempted, the solver is called by the CS-Prolog engine to verify whether the current set of constraints would remain satisfiable after the unification. If not, the unification attempt fails. (This kind of unification is handled by the solver as a special constraint.)

The ML solver accepts structures as constraints where the main functor is one of $(=:=)$ **/2**, $(=<)$ **/2**, or $(>=)$ **/2**, and the arguments of the structure are *CLP linear expressions*, i.e. extended arithmetic expressions that may contain unbound variables and constrained variables, but only in such a way that after 'flattening' the expression and evaluating the variable-free subexpressions, the result is a (possibly multi-variable) polynomial, each addend of which has a summary degree of at most one. These structures represent arithmetic equalities and (non-strict) arithmetic inequalities; the interpretation of the main functor is the same as for arithmetic evaluation.

The solver uses *arithmetic typing* when the arithmetic equality is evaluated, in accord with the usual meaning of $(=:=)$ **/2**. Constrained variables can be unified with both integers and floating points (if the current set of constraints remains satisfiable after the unification).

Changes made to the current set of constraints are backtrackable: any constraint added to the set during a call is removed when backtrack over that call is performed.

Several interface predicates accept a class of structures called *CLP evaluable linear expressions* as one of their arguments. These are similar to CLP *linear expressions*, with the difference that they cannot contain unbound variables.

Errors

Some error exceptions can be reported by the solver due to internal conditions for almost any interface predicate call, and also for other calls attempting unification where constrained variables are involved. The general form of these exceptions is the following:

`clp_system_error`

Other_info contains a list of the form `[internal_error,ErrCode]`, where **ErrCode** is an integer code specifying the particular error detected

These errors either occur because of exceeding an internal limitation, or are symptoms of an internal programming error detected by the solver's auto-diagnostic component itself.

2. The 'normal' interface predicates

The ML solver supports all normal CLP interface predicates, and no other interface predicates are defined for it. In this chapter the specific behavior of these predicates are described. There is some repetition in the narrative text so that the basic functionality could be learned without referencing the corresponding chapter of the User's Manual, but only the specific error messages are included here.

clp_constraint/1

Description

`clp_constraint(ConstrList)`

This is the central element of the CLP interface, used for defining constraints for the currently selected solver.

ConstrList is a list of structures (specific for the solver involved), each structure representing one constraint. The ML linear solver accepts structures where the main functor is one of (=:=) **/2**, (=<) **/2**, or (>=) **/2**, and the arguments of each structure are *CLP linear expressions.*

The solver analyzes the constraints contained in **ConstrList**, decides whether they are syntactically correct and if so, whether adding them to the current set of constraints yields a consistent state.

If any error is detected, then an appropriate error is raised.

Otherwise, if the new set of constraints is inconsistent with the current status, then the call fails.

If no error is detected and the new constraints are accepted, then the call succeeds. Unbound variables encountered in the new constraints become constrained variables (corresponding to new problem variables).

The predicate is backtrackable. After backtracking over the call the status of the model maintained by the selected solver instance reverts to the status that was in effect before the call.

Template and modes

`clp_constraint(+list)`

Errors

`domain_error(clp_relation_functor,Constr)`
The main functor of constraint **Constr** occurring in **ConstrList** is not one of the relation functors accepted by the solver. **Other_info** contains the indicator of the offending functor.

`domain_error(clp_linear_expression,Constr)`
One of the structure arguments in constraint **Constr** occurring in **ConstrList** does not comply with the definition of *CLP linear expression.*

`domain_error(proper_list,ConstrList)`
ConstrList is either an open list (ending with an unbound variable) or an improper list (ending with a term other then **nil**).

clp_type/[2,3]

Description

```
clp_type(Expr, Type)
```

is equivalent with

```
clp_type(Expr, Type, _)
```

```
clp_type(Expr, Type, Extra)
```

Expr should be a *CLP evaluable linear expression* (not containing unbound variables). The solver qualifies the expression (based on the current status of the constrained variables occurring in it), and returns an atom representing the result of the classification. The returned value is unified with **Type**.

The ML solver returns one of the following atoms for **Type**: **free**, **lobnd**, **upbnd**, **bounded**, **fix**, **number**, where **number** means that **Expr** is a fully evaluable arithmetic expression (with constant value), the other categories correspond to value ranges like the ones described for the possible states of constrained variables (see the chapter on CLP extension in the User's Manual).

The solver at present unifies **Extra** with **nil**.

Template and modes

```
clp_type(+term, ?atom)
```

```
clp_type(+term, ?atom, ?List)
```

Errors

```
domain_error(clp_evaluable_linear_expression,Expr)
```
Expr does not comply with the definition of *CLP evaluable linear expression*.

clp_max/[2,4]

Description

```
clp_max(Expr, Value)
```

is equivalent with

```
clp_max(Expr, Value, [], _)
```

```
clp_max(Expr, Value, Query, Answer)
```

Expr should be a *CLP evaluable linear expression* (not containing unbound variables). The solver attempts to calculate the maximal value that the expression can assume subject to the current set of constraints. If the maximum does not exist (the expression has no upper bound) then the call fails, otherwise **Value** is unified with the calculated maximal value. The **Query** argument can contain solver-specific query items (one item, or a list of items) about the solution of the current set of constraints corresponding to the maximum found. **Answer** is unified with the item, or with a list of items, that supply the answers to the query item(s) contained in **Query** (see also **clp_value/2**).

The ML solver at present ignores **Query** and unifies **Answer** with **nil**.

Template and modes

```
clp_max(+term, ?number)
```

```
clp_max(+term, ?number, +term, -term)
```

Errors

```
domain_error(clp_evaluable_linear_expression,Expr)
```
Expr does not comply with the definition of *CLP evaluable linear expression*.

clp_min/[2,4]

Description

These predicates are essentially the same as **clp_max/[2,4]** above, considering the equivalence

$$\min \{ f(x) \} \ == \ - \max \{ -f(x) \}$$

Template and modes

```
clp_min(+term, ?number)
clp_min(+term, ?number, +term, -term)
```

Errors

```
domain_error(clp_evaluable_linear_expression,Expr)
```
Expr does not comply with the definition of *CLP evaluable linear expression*.

clp_value/2

Description

```
clp_value(Query, Answer)
```

Query is a *CLP evaluable linear expression* (not containing unbound variables), or a list of such expressions. If **Query** is a structure (one expression), then the solver evaluates the expression, substituting values for the constrained variables in the expression from the feasible solution maintained as part of the current state, and unifies **Answer** with the result of this evaluation.

If **Query** is a list of expressions then the solver builds a corresponding list of results evaluating each expression as in the previous case, and unifies **Answer** with this list.

The ML solver at present accepts only single constrained variables or fully evaluable arithmetic expressions as **Query** items.

Template and modes

```
clp_value(+struct_or_list, ?number_or_list)
```

Errors

```
type_error(clp_evaluable_expression,Query)
```
Query is an atom.
```
domain_error(clp_evaluable_linear_expression,Constr)
```
Query does not comply with the definition of *CLP evaluable linear expression*.
```
clp_system_error
```
Query is a list containing too many (more than 1000) items. **Other_info** contains the **atom `list_is_too_long`**.

clp_debug_mode/1

Description

```
clp_debug_mode(Flags)
```

Passes the value of **Flags** to the selected solver. The intent of this predicate is to give the user program some control over any debugging facility provided by the specific solver.

The ML solver interprets the value of **Flags** as a set of flag bits. There is an interactive debugging feature that allows the user to investigate different aspects of the internal state of the solver. Most of the defined flag bits are assigned to certain events during the solver's execution, and if the particular flag bit is *on*, then the debugger is activated before and/or after the event is processed.

Flag bit value	Event
1	New problem variable created (only 'after').
2	Consistency check when new constrains are added explicitly ('before' and 'after').
4	Consistency check for unification ('before' and 'after').
8	Backtracking ('before' and 'after').
64	(used during tests only, activates certain prepared printouts).
128	Prints the labels for all trace points, but does not activate the debugger if the specific flag corresponding to the trace point is not set.

Flag bits not defined in the table above are ignored by the debugger facility.

Template and modes

```
clp_debug_mode(+non_negative_integer)
```

Errors

No specific error

3. Limitations and known errors

The ML solver can be active only in one CS-Prolog process at a time. It is activated implicitly when the first 'significant' interface predicate call is issued by the process. (All normal interface predicates are deemed significant except **clp_debug/1**, though normally **clp_constraint/1** is the one called first.) Since no method is provided for explicitly deactivating a solver, the process using the solver must be terminated before another process can activate it again.

The maximal number of internal variables maintained by the solver (*problem variables* and *slack variables* together) at present is limited around 130.

The solver does not comply with the requirement that all normal predicates must be fully backtrackable. It maintains a 'feasible solution' all the time as part of its internal state, but this is not reverted during backtracking over querying calls. **clp_value** calls are evaluated for some simple queries against the current feasible solution, in other cases a new feasible solution is located for composing the answer. Furthermore, in the general case clp_type, clp_min, and clp_max all change the current feasible solution. The problem is that these changes are not undone either immediately or on backtrack, so e.g. a clp_value can yield different results depending on the execution history, when it should provide identical results.