

CS-Prolog II

The CLP Solver Developer's Guide

ML Consulting and Computing Ltd.

Budapest, Hungary

January, 1999

Contents

1. Introduction.....	5
1.1 Problems with the added term type	7
1.2 Problems and open questions pertaining to 'normal' CLP interface predicates	7
2. The solver interface.....	8
2.1 Organization of backtracking.....	9
2.2 Entry points into a solver, exposed via the interface table	10
2.3 Callback functions (entries in the 'XpClpCallbacks' table).....	15
2.4 Support functions for arithmetic evaluation ("arithm.h")	16
3. Programming conventions	20
4. Macro definitions for compiling the sources	21

1. Introduction

The description of the general CLP extension model for CS-Prolog and the set of system-provided CLP interface predicates are now included in the CS-Prolog II User's manual as two separate chapters (*The Constraint Logic Programming (CLP) extension* and *Predicates for the CLP extension*) and in a Supplement (*The ML Linear Solver*). The corresponding parts are mostly removed from this document in order to simplify updating. Some parts, however, are repeated here with added internal details that concern only developers, or explanation of terminology, or just to help continuous reading. In case of ambiguity, the User's Manual is to be considered as more up-to-date.

The CS-Prolog runtime program can be configured to include one or several different CLP solvers (the technical maximum at present is four solvers, but this is only a matter of allocating a bit-field of appropriate width within an internal structure).

For the sake of simplicity, when describing the interaction between different parts of the system, we shall speak of 'the solver' and 'the core', meaning the subsystem that is responsible for maintaining and repeatedly re-evaluating the set of constraining conditions, on one hand, and the component that keeps track of the active instances of the different solvers, dispatches requests originated by the Prolog program to the appropriate instance, and performs other system-related tasks, on the other hand.

The CLP-related built-in predicates provided by the system can be divided into three groups. The first group is concerned with the term type system extension, and contains the following predicates:

```
constrained_var(?term)      % c.f. var(?term)
strict_nonvar(?term)       % c.f. nonvar(?term)
strict_ground(?term)       % c.f. ground(?term)
```

The second group consists of the predicates used directly in the work with the solvers:

```
query_clp_config/4
select_clp_solver/[0,1]
```

The third group consists of the predicates used directly in the work with the solvers (the 'normal' CLP interface predicates):

```
clp_constraint/1,
clp_type/[2,3]
clp_max/[2,4]
clp_min/[2,4]
clp_value/2,
clp_debug_mode/1,
```

The first two groups are entirely implemented by the core system; the predicates in the third group require cooperation between the core system and the particular solver.

The Prolog interface for a solver may consist of some or all of the 'normal' predicates and the set also may be extended by defining specific predicates as 'foreign' ones. In fact, the solver can entirely ignore the 'normal' predicates. (Using 'foreign' interface is just a bit more complicated.)

Each solver has a **SolverId** assigned to it by the configuration module. This **SolverId** is passed to the particular solver during the one-time initialization call, before the execution of the user program begins. The solver, which is, assigned the value 0 as **SolverId** is called the default solver; it is selected initially and can be selected by calling **select_clp_solver/0** (without argument). (The default solver is described by the first entry in the configuration table.)

The actual properties of the solvers are specified in the configuration table; in the most general case several CS-Prolog processes can have instances of more than one solver running (but only one instance of each

particular solver), and each solver can be active in more than one CS-Prolog process (multithreading), and can 'live together' with instances of other solvers in one thread. (Note that here and in the sequel, 'thread' is used as a synonym for 'CS-Prolog process' for short, not for the operating system's thread concept.) Different instances of the same solver should be unrelated in the sense that problem-specific data from one thread should not be manipulated by an instance living on another thread.

The supposed general behavior of a solver instance is as follows.

The instance is initialized when the first 'significant' predicate call is issued by the user program in the thread. (If the solver defines its own interface predicates, then it must notify the core system and so trigger the initialization call from it). The instance starts with an empty 'model' (system of constraints). During 'forward' execution, new constraints are incrementally added to the model. The solver evaluates the resulting constraint set, and, if it proves feasible, accepts the additions (the call succeeds), otherwise rejects them (fails). If the predicate that passes the new constraint - typically `clp_constraint` - succeeds then all unbound variables occurring in the passed constraints must be transformed into *constrained variables*. There is an internal interface function for this purpose. If the Prolog program later performs backtrack over a **clp_constraint** call, the solver must revert to the state that was in effect before that call. Removal of problem variables from the internal state during backtrack also must be reported to the core system.

The core system can assist in this backtracking by calling a hook function if the solver had requested this. If the solver uses its own interface predicate for constraint addition, then it must organize backtracking itself by using the functions provided as part of the 'foreign' implementation set (see 'csprolog.h').

Beside **clp_constraint** or its solver-specific counterpart, adding special constraints equivalent with the explicit form

```
<constrained_variable> ::= <value>
```

or

```
<constrained_variable> = <value>
```

can also be originated implicitly from Prolog unification (the actual form depends on which relation is understood by the solver). From the solver's point of view this case is not much different from the explicit predicate call, except that the calling environment is different (the set of usable 'foreign' interface functions is restricted), and that backtracking in this case will always be organized by the core.

It is important, however, that the 'observable' state of the model after backtracking should be identical with the state before the call the effect of which had just been undone by the backtrack procedure. (Internal details exposed by **clp_debug** are not considered as part of the observable state).

The rest of the interface predicates serve only for querying; they should not change the actual state of the model. (The ML solver does not comply with this requirement - after **clp_max** or **clp_min** the observable state changes, i.e., the result from **clp_value** is different).

The **clp_constraint/1** predicate takes as argument 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 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.

NOTE:

As stated in the description of the extended unification rules for constrained variables, the solver never binds a constrained variable to numeric value directly because in the general case the exact type is not determined. With strict typing, we can consider automatic binding, but the consequences must be investigated first.

1.1 Problems with the added term type

The introduction of additional type-checking predicates, as done presently, is an imperfect way of handling the problem of types. Someone has to analyze the question theoretically.

When a term containing constrained variables is ‘output’ (e.g. using **write_canonical**) and then read back, the new term will not be equivalent with the original (although can be unified with it). No renaming of variables can make the two terms equivalent as required by the standard.

Backtrackable Prolog database updating operations should retain constrained variables like **set_global_value_b/2** does, but in order to implement this behavior we would have to completely revise the compiler, so it is not done.

Although I have scanned the implementation of built-in predicates that are written in C, and corrected many of them so that they treat constrained variables according to their specification, I know that this task is not completed. The problem is that there are local optimizations based on those assumptions about the term type system that are not valid with the extension. In particular, I have not touched the external database interface, and, as a specific example, I know that **subatom/3** needs refurbishing.

The solution-collecting predicates are completely baffled by constrained variables. I had no time even to think about how they should treat them.

1.2 Problems and open questions pertaining to ‘normal’ CLP interface predicates

The problems are centered on the notion of ‘observable state’. It is tacitly understood in the specification of predicates above that the solver always maintains one particular feasible solution consistent with the current set of constraint, and all queries are answered using values for constrained variables from this solution. This, however, might put an unnecessary burden on the solver (the ML solver, for example, does not comply with this requirement).

I can see two ways for simplification:

1. In ‘clp_value’ we can lift the requirement that all answers pertain to the same solution, and simply tell that each answer corresponds to some feasible solution, not necessarily the same one.
2. The optional 3rd and 4th arguments of ‘clp_min’ and ‘clp_max’ have been introduced based on the supposition that the components of the solution corresponding to the extremal value found are available only during the call; after the call the state of the solver reverts to the maintained feasible solution. This behavior can be redefined so that ‘clp_min’ and ‘clp_max’ drive the solver into a new state where the maintained feasible solution corresponds to the extremum found. If this approach is followed then there is no need for the optional arguments; the components of the solution can be queried using subsequent ‘clp_value’ calls (but in this case the individual answers must be consistent with each other).

2. The solver interface

The solver must expose for the runtime system (the 'core') a table containing function pointers to specific functions (entry points). The structure of this table is defined in the 'clp_if.h' header file. The table itself must be globally visible (for the link editor); i.e. it must have an 'external' symbol for its name. This name is used in the configuration module (initializes a field in the table entry for the solver).

The core on its part also exposes a similar table, also described and declared in the 'clp_if.h' header. This table has the fixed name **XxpClpCallbacks**. The callback functions contained in this table can (or should) be called by the solver in specific circumstances.

In addition to the callback functions, the solver can use the services of the runtime system exposed through 'csprolog.h' for implementing 'foreign' predicates - not necessarily for that purpose. There are also functions specifically exposed for the benefit of solvers through the header file 'arithm.h'. Note that arithm.h exposes more than really necessary, especially by including another header file 'xsp_uten.h', simply because it was not designed for exposure.

'csprolog.h' is explained in the User's Manual (chapter 'The C Interface' - an update is being prepared). The other two will be shortly described in the following.

In the course of interaction between the core and a solver, in most cases the core has to know which solver is being served (the actual instance is determined by the thread currently running). When the interaction is originated by one of the 'normal' predicate calls, then the core itself dispatches the request to the currently selected solver, so the solver is known. Similarly, when a 'core-assisted' backtrack call is dispatched to the requesting solver, the solver's identity is also known.

However, if the solver obtains the control bypassing the core, via a 'foreign' predicate or during backtrack organized by itself, the core is ignorant about the identity of the solver requiring services, so this information must be communicated to it.

Part of the service requesting callbacks is self-describing in this respect: the SolverId is explicitly included in the argument list. Others, however, rely on the status information maintained by the core. When control is obtained independently, then the solver has to announce itself before requesting any such service. A pair of interface functions, **sign_on** and **sign_off** is introduced for this purpose. (The asynchronous **reset** service also relies on the identity of the currently running solver instance.)

In the description of the interface functions that follows, we shall use the structure item names for the corresponding function name, the precise C syntax being somewhat clumsy. The actual function names supplied by the parties will be different.

Some of the functions should return a value of type **CspRetC**. The interpretation of this value is explained in detail in the User's Manual, in chapter "The C Interface"; here is a brief summary:

There are two special values, **XXP_E_SUCCEED** and **XXP_E_FAIL**, that are considered both as 'normal' outcome of the function call. Any other value returned will be treated as an indication of some error or extraordinary condition. Such exceptional values are usually prepared by calling one of the several 'xsp_error' functions listed in 'csprolog.h' for different error categories, which prepare additional data for reporting the situation.

The specialty of the interface functions is that they do not return this return code immediately to the interpreter as the outcome of the predicate call, but rather to the CLP dispatcher within the core as intermediate result. For the dispatcher, **XXP_E_SUCCEED** means that it should go on performing the actions necessary for serving the current predicate call. Any other returned value causes the dispatcher to finish the current course of actions in the shortest possible way and report the condition (failure or exception) to the interpreter.

In order to support mixed integer/float arithmetic we've defined a union type **XxpNumType** for handling these values (see "arithmh"). The discriminator field, named 'num_flag', must be set to either **XXP_T_INT** (when the structure holds a [long] integer - in the pseudo-field 'int_value') or to **XXP_T_FLOAT** (when the structure holds a **CspFloatType** [double] - in 'float_value'). There is a short field named 'user_data', unused at present (in an earlier version we needed it); now it is available for any good purpose.

A note on terminology:

After some hesitation, we selected the term ‘constrained variable’ for the special object introduced, but in some function names and other definitions the previous choice of ‘restricted variable’ is still present. Don’t be confused, they are the same. Also, in some places ‘constrained variable’ is abbreviated as ‘c-var’ or ‘CVAR’.

2.1 Organization of backtracking

The state of the model maintained by a solver instance must be synchronized with the state of the CS-Prolog evaluation stack for the host thread. This means that any change in the model caused by the evaluation of an interface predicate or a unification involving constrained variables must be ‘undone’ when the interpretation backtracks over the predicate that originated the change. The CS-Prolog interpreter maintains a special data area, called ‘trail’, for this purpose. This area is used in the first place by the interpreter itself for registering variable bindings that should be undone during backtrack, but special entries can also be inserted on behalf of the implementation of any built-in predicate that requires it. (For more details, see “The C Interface” in the User’s Manual).

The core offers a facility that can make the organization for backtracking easier than the general way. It has only one drawback: only one ULONG item is saved as user data in the trail entries for identifying the changes to be undone. In most cases, however, this is sufficient, because the solver will maintain its own internal stack, and the data item returned from the trail will be used only for identifying the appropriate stack level (or just for checking that synchronism is still correct).

The recommended practice for any built-in predicate (and also for solver interface functions which are in effect parts of the implementation of a built-in predicate) to perform all validity checking before any change is effected in their internal state or in the environment. When this practice is followed faithfully, then trailing is needed only at the very end of a successful execution. The problem is that it is not always easy to keep the phases separated, because it would require the duplication of the components affected by the changes. In such cases, it is acceptable to perform the changes gradually, by stages, and make a trail note for each stage. The number of stages is still to be held at a practical minimum in order not to overtax the trail, which is the least recoverable resource of the interpreter.

(Garbage collection cannot reclaim ‘dead’ entries from it, only effective backtracking will free the part used in the backtrack process.)

If the solver is willing to rely on the core-assisted backtrack service, it can set up trail notes by calling the **put_data_on_trail** callback function. Several interface functions, as e.g. **constraint**, can also request final trail note through the special output arguments provided specifically for this purpose. The advantage of the latter method is that the core can combine its own backtrack data with the solver’s data in one trail note.

The special output arguments for requesting ‘final’ trail note are represented by the same formal parameter names in all function prototypes where this facility is provided (the last two arguments), as in the following example:

```
CspRetC constraint(xxp_cell Constr, Boolean *TrailingIndPt,  
                 ULONG *TrailDataPt)
```

TrailingIndPt is a pointer to a Boolean variable. If the called function (in this case **constraint**) places TRUE (or any other non-zero value) into this variable, and passes XXP_E_SUCCEED to the caller as the return value of the function, then the caller (some part of the core) will set up a trail note for the current interpretation status, and stores the value placed by the function into the ULONG pointed at by **TrailDataPt** (or sets up a combined trail note, used also for its own purpose).

Later, if backtracking over this point occurs, the core will call the entry point **backtrack** provided by the solver, passing the saved ULONG value as the sole argument.

Notes:

1. The trail request will be honored only when the called function indicates success by returning `XXP_E_SUCCEEDED`. For any other return value an immediate backtrack follows without setting up the final trail note.
2. The Boolean variable pointed at by the output argument **TrailingIndPt** is initialized to `FALSE` as default value before calling the function. The `ULONG` pointed at by **TrailDataPt** is not initialized.
3. If the solver does not intend to use the core-supported backtracking at all, then the **backtrack** entry in the interface table can be set to `NULL`. In this case, any trailing request to the core (**put_data_on_trail** call or implicit final trail request via the mechanism described here) will cause an error exception.

2.2 Entry points into a solver, exposed via the interface table

I. System initialization/termination

```
int init_package(unsigned SolverId)
```

This is the 'one-time' initialization function of the solver. It is called in the initial phase of the program execution, before the Prolog program execution actually begins. **SolverId** is the solver identifier value assigned to the solver by the CLP configuration module. The solver should save this value; it will need it as argument in several calls.

The function should return 0 if the initialization is successful. Any other value returned will lead to immediate termination of the runtime system, with a generic error message. If the solver has additional information about the error then it can write it out directly to **stderr**.

```
void terminate_package(void)
```

This is the counterpart of **init_package**. Called immediately before the application exits; gives the opportunity for the solver to release any resources held, write out statistics, etc.

II. Multithreading support

If the solver supports multithreading, it needs separate data area for each thread where it has an active instance. The core offers assistance in creating new instances and in switching from one thread-specific area to the other ('swapping') when necessary.

The solver might prefer to hold some frequently needed data items and pointers in a compiler-allocated static area, thereby eliminating one level of indirection when accessing these items. The static area is saved to a thread-specific shadow area when needed and is reloaded from the shadow area of the new thread when thread switching occurs. The assistance for swapping is designed to help this kind of organization.

An important aspect of the swapping strategy is that solvers do not need strict synchronization with the current Prolog process, because not all Prolog processes are associated with an active instance of a particular solver. If a solver has only one active instance then it won't need swapping at all.

```
void *init_instance(long DebugFlags)
```

This entry point is called immediately before the core is going to dispatch the first 'significant' call to the solver and there is no active instance of the solver in the current thread, or at the request of the

solver itself when it obtained control independently of the core and decides that it needs initialization of a new instance (see the **wake_instance** callback later).

DebugFlags is the argument saved from the latest 'clp_debug' predicate call (which does not cause instance initialization). If there was no such call in the current thread before the instance initialization then the value 0 is passed.

The solver should return a (**void ***) type value. The core does not interpret this value, simply saves it for the current instance, and presents it as one argument of a later swap request, if such will occur at all. The solver has full freedom in choosing this value, but the original intent is to return one of the following:

- If the solver uses static data area and does not allocate a shadow area immediately, then it can return NULL. For non-multithreading solvers no swap request will occur; otherwise when the first swap request comes and presents this NULL value, the solver can allocate an appropriate shadow area, save the content of the static area into it, and pass the address of the shadow area as the result from the swap request.
- If the solver immediately allocates a thread-specific area (either as shadow area for the static area, or for direct use), then it should return the address of the allocated area, and use it without modification as the ensuing swap requests present it.

Note: if **init_instance** is called because the core detected the 'first significant predicate call', it is not sure that an interface entry corresponding to that predicate will follow immediately. The core might reject the predicate call at a later phase, before the solver gets involved.

```
void terminate_instance(void *user_area_pt)
```

This is the counterpart of **init_instance**, with an important difference. **init_instance** is always called within the execution of the thread for which the new instance is to be installed; **terminate_instance** might be called when another thread is current. The **user_area_pt** argument contains the value saved from the latest preceding **swap** or **init_instance** call. The solver is expected to free any thread-dependent resources (memory areas allocated). The thread associated with **user_area_pt** will not be activated any more after this call.

```
void *swap(void *SaveToPt, void *LoadFromPt)
```

This entry point is called by the core when it determines that the current instance of the solver is to yield control to another instance (of the same solver - different solvers do not interfere with each other). This can happen in two different situations. The first is when the Prolog process scheduler switches to a thread where there is a live instance of the solver, but up to this moment, another instance has been controlling the solver's resources. The other situation is when a new instance is to be started in the current thread, and (as in the previous case) an other instance is controlling the solver's resources.

In both cases, the **SaveToPt** argument contains the value saved from the last **swap** or **init_instance** request for the instance being replaced. If it is NULL (only when saved from **init_instance**) then the solver should allocate memory area for saving any necessary data for the instance being replaced, otherwise it can use the area pointed at by **SaveToPt** (it can also reallocate the area if necessary). In any case, the thread-dependent data must be saved, and the address of the current save area must be passed back to the core as return value. (If the solver accesses all thread-dependent data through this pointer, then nothing needs to be saved.) The return value from **swap** cannot be NULL. To state it in one sentence: **swap** offers the possibility for the solver on each occasion to change the save area for the instance being replaced, and stores the address of the new area, which, however, must not be NULL.

If **LoadFromPt** contains NULL, this indicates that a new instance is to be initialized, i.e., an **init_instance** call will follow immediately. In this case, the solver is supposed to prepare any common resources for accommodating a new instance. Otherwise, when **LoadFromPt** is not NULL, then we

face the situation of thread switching, and **LoadFromPt** contains the value saved for the instance that is to be activated. After saving all necessary data for the old instance, the solver is expected to fetch all data that needs reloading from this area.

For a non-multithreading solver, this table entry must be set to NULL.

III. Support for 'normal' interface predicates

If the solver does not support some of the interface predicates listed in this part, the corresponding entry in the interface table should be set to NULL.

```
CspRetC constraint(xxp_cell Constr, Boolean *TrailingIndPt,
                  ULONG *TrailDataPt)
```

This entry is called when the core encounters a **clp_constraint** call, finds it valid as far as its competence goes, and any necessary preparations are already performed (swap request, instance initialization).

Constr contains the (pre-checked) argument from the call of the original **clp_constraint** predicate. Pre-checking ensures that **Constr** is a non-empty list (empty list is simply accepted by the dispatcher and success is indicated). In the case of solvers qualified as 'xenophobic' in the CLP system configuration module, it is also ensured that **Constr** does not contain any 'alien' constrained variables belonging to another solver. Note, however, that pre-checking does not filter out improper lists (the final element of which is not **nil**).

The solver is expected to analyze the constraints contained in **Constr**, decide 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 by the function, then an appropriate error code should be returned.

Otherwise, if the new set of constraints is inconsistent with the current set, then the call should return **XXP_E_FAIL**.

If no error is detected and the new constraints are accepted, then **XXP_E_SUCCEED** is to be returned, and the solver has the option of requesting the dispatcher to set up a 'final' trail note and pass the **ULONG** value to be saved in that trail note. Intermediate trail notes should be set up by calling **put_data_on_trail** at the appropriate points. (See 'Organization of backtracking' earlier).

Unbound variables encountered in the new constraints are going to become constrained variables (corresponding to new problem variables) if the call succeeds. This act requires careful organization. The factors to be considered are the following:

When a variable is transformed (by the **mk_restr_var** callback function), then all instances of the variable occurring in the input constraint set (**Constr**) become associated with the problem variable (i.e. all instances are transformed at the same time).

- Usually it is impossible to delay this transformation until the success of the call is ensured, so preparations for undoing the effect are required, preferably without involving the trailing mechanism (for efficiency's sake).
- Further complication is caused by the fact that the solver might lose control without explicitly returning, due to some exception raised directly inside a called support function (a rare event, but not impossible). The asynchronous **reset** service is provided to cope with such events (for undoing unfinished and untraced transactions).

It is supposed that the problem variables maintained by the solver are identified by some small non-negative integers (index values). This value must be less than **CLP_MAX_SOLVER_VARS** (defined

in “`clp_if.h`”), and the total number of such variables must not exceed `CLP_MAX_PROBLEM_VARS` (op. cit.).

‘Undoing’ the creation of new constrained variables is performed by calling the **`remove_last_restr_vars`** callback function. The use of this function must be strictly synchronized with the creation of those variables: it can be called only immediately before or after backtracking over the point where the variables concerned had been created. (Remember that raising an exception also involves backtracking.)

```
CspRetC type(xxp_cell Expr, xxp_cell *TypeCellPt, Boolean NeedsExt,
             xxp_cell *ExtCellPt, Boolean *TrailingIndPt,
             ULONG *TrailDataPt)
```

This function is called for the **`clp_type`** predicate. **`Expr`** is the pre-checked expression the type of which is to be determined. Pre-checking ensures that **`Expr`** is a structure, or a constrained variable, or a number, and, for ‘xenophobic’ solvers, that it does not contain ‘alien’ constrained variables.

The result of the evaluation should be a solver-dependent atom (as: ‘**`free`**’, ‘**`lobound`**’, ‘**`fixed`**’, etc.), placed in **`*TypeCellPt`**. Argument **`NeedsExt`** indicates whether or not additional information is requested by the caller. Additional data (solver-dependent term, preferably list) should be placed into **`*ExtCellPt`**. For the last two arguments see ‘Organization of backtracking’ earlier.

```
CspRetC value(xxp_cell Arg1, xxp_cell Arg2, Boolean *TrailingIndPt,
             ULONG *TrailDataPt)
```

This function is called for the **`clp_value`** predicate. **`Arg1`** is the pre-checked first argument of the predicate call. Pre-checking ensures that **`Arg1`** is a structure or a non-empty list or a constrained variable (numbers and the empty list are handled directly by the dispatcher), and in case of ‘xenophobic’ solver, that it contains no alien constrained variable. **`Arg2`** is the second argument of the predicate call (for the result), unchecked.

The reason why **`Arg2`** is passed directly instead of an `xxp_cell` pointer for the constructed result is that some local optimization is possible when treating e.g. a (partially) filled answer list. For the last two arguments see ‘Organization of backtracking’ earlier.

```
CspRetC max_or_min(xxp_cell Expr, XxpNumType *ValuePt, Boolean IsMax,
                  Boolean NeedsExt, xxp_cell *ExtCellPt,
                  Boolean *TrailingIndPt, ULONG *TrailDataPt)
```

This function is called for the **`clp_max`** and **`clp_min`** predicates; argument **`IsMax`** indicates which one exactly (TRUE for **`clp_max`**, FALSE for **`clp_min`**).

`Expr` is the pre-checked first argument of the original predicate call. Pre-checking ensures that **`Expr`** is a structure, a number, or a constrained variable, and, for xenophobic solvers, that it does not contain ‘alien’ constrained variables.

The calculated extremal value is to be placed into **`*ValuePt`**.

The role of the remaining arguments is the same as for **`type`**.

```
void debug_mode(long Flags)
```

This function is called for the **`clp_debug_mode`** predicate if the solver is already active. (‘Early’ calls are intercepted by the dispatcher, and the last **`Flags`** value is passed to the solver at instance initialization time.) **`Flags`** is the value of the (non-negative) Prolog integer argument of the predicate call.

IV. Miscellanea

```
CspRetC unification(xxp_cell Term1, xxp_cell Term2,
                   Boolean *TrailingIndPt, ULONG *TrailDataPt)
```

This function is called directly by the CS-Prolog engine when unification of a constrained variable owned by the solver is attempted either with another such constrained variable, or with a number when the status of the constrained variable is not FIXED.

Term1 is always a constrained variable, **Term2** is the other term (number or constrained variable). If both are c-vars, and at least one of them is not FIXED, then **Term1** is not FIXED (**Term2** cannot be 'less-determined' than **Term1**). It is supposed that the solver treats this call as a special case of a new constraint received. If **Term2** is a number and the function returns success indication, then the c-var in **Term1** becomes (conceptually) bound to the number.

It must be considered that this function is called from a more restricted environment than the others are, so resource-intensive operations and other extravagant functions are to be avoided. (The exact rules are not established - sorry. The main example of 'extravagant' function is calling Prolog from C.)

For the last two arguments see 'Organization of backtracking' earlier.

```
void backtrack(ULONG TrailData)
```

This function is called when the interpretation backtracks over a point for which a core-supported trail note has been set up for the solver (see 'Organization of backtracking' earlier). **TrailData** is the value stored in the corresponding trail note.

The environment from which this function is called is also restricted, like that for **unification**.

```
void reset(void)
```

This function is provided for coping with the problem of losing the control unexpectedly due to an internal exception during the elaboration of some function call with side effects (see also the description of **constraint** for details).

The function is called during the handling of a raised exception, immediately before starting the replacement call (after all necessary backtracking), if

- (1) The solver did not inhibit this service by specifying NULL in the table entry, and
- (2) The dispatcher is aware of the fact that the solver had been 'hit' by the exception, i.e. the exception occurred while the solver had been performing some action. See the introductory part to the solver interface description for details about when does the dispatcher know the active solver.

This function can perform any necessary clean-up actions for which preparations had been made in advance.

The environment from which this function is called is also restricted, like that for **unification**.

```
Boolean is_rst_var_fixed(unsigned Index, XxpNumType *ValuePt)
```

This function is called by the dispatcher during special unification processing; it requests information about the status of the problem variable identified by **Index**. If the status is FIXED then the function should return TRUE, and place that fixed value into ***ValuePt**. Otherwise, the function should return FALSE.

```
Boolean strict_numeric_types(void)
```

This function informs the dispatcher about a static attribute of the solver, namely whether ‘Prolog’ or ‘arithmetic’ typing is used for numeric values (see earlier). The return value should be TRUE for ‘Prolog’ typing and FALSE for ‘arithmetic’ typing.

```
char const *description(void)
```

This function should return a null-terminated character string containing a BRIEF description of the solver (copyright, version, product name, etc.). At present, this function is called when the ‘-ver’ command line option is specified for running a program (to display version data on stderr).

2.3 Callback functions (entries in the ‘XpClpCallbacks’ table)

```
void mk_restr_var(xxp_cell VarCell, unsigned Sid, unsigned Index)
```

This function should be called when an unbound variable occurring in a new constraint is to be transformed into a constrained variable associated with a problem variable (newly created by the solver - see **constraint** for details). **VarCell** is the variable in question, **Sid** is the solver ID of the solver, and **Index** is the identifier of the problem variable assigned by the solver. **Index** must be less than CLP_MAX_SOLVER_VARS, and the total number of ‘live’ problem variables must not exceed CLP_MAX_PROBLEM_VARS.

```
void remove_last_restr_vars(unsigned Sid, unsigned NVars)
```

This function should be called when - during backtrack or internal undo processing - some problem variables (already associated with constrained variables) are discarded by the solver. **Sid** is the solver ID of the solver, **NVars** is the number of problem variables discarded in one batch. Note that only the ‘newest’ problem variables can be discarded (in reverse order compared to the corresponding **mk_restr_var** calls). See also **constraint** for more details.

Important note: the solver is not allowed to discard problem variables at will after creation. Constrained variables associated with a particular problem variable will be ‘unbound’ only by the interpreter during backtrack; discarding the problem variable must be as tightly synchronized with this event as possible.

```
unsigned restr_var_ind(xxp_cell CvarCell, unsigned Sid)
```

This function returns the identifier (index) of a problem variable associated with the constrained variable **CvarCell**. **Sid** is the solver ID of the solver owning the problem variable.

If **CvarCell** is not a constrained variable, then an exception is signaled directly (the control will not return to the caller). If **CvarCell** is an ‘alien’ c-var (for **Sid**), then the special value CLP_ALIEN_VAR_IND is returned (defined in “clp_if.h”).

```
void sign_on(unsigned Sid)
```

This function should be called if the solver obtains the control bypassing the dispatcher, if any services requiring that the dispatcher know about the identity of the active solver, are going to be used (including the asynchronous **reset** service). See the introduction to the solver interface for details.

sign_on must not be called when the control is obtained from the dispatcher (such call causes an error exception). **sign_on** also must not be called before the initialization of the solver instance (see **wake_instance** later).

```
void sign_off(unsigned Sid)
```

This function must be used in tandem with **sign_on** above. It is an error (not always detected) if any top-level function returns control to the system while the solver is 'signed on'. In the case when the solver loses control because of exception, the missing **sign_off** is supplied by the system (actually, it is exactly what triggers the **reset** call).

sign_on/sign_off pairs cannot be nested!

```
CspRetC wake_instance(unsigned Sid)
```

This function should be called when the solver receives the control via a foreign predicate call, bypassing the dispatcher, and there is a possibility that the instance of the solver for this thread is not initialized yet. **Sid** is the solver ID of the solver. The dispatcher for this call checks the status of the instance. If it is already initialized, then simply returns with **XXP_E_SUCCEED**. Otherwise, the dispatcher performs the necessary initialization, in the course of which the solver's **init_instance** entry point is also called.

```
void put_data_on_trail(ULONG TrailData)
```

This function can be used as a simplified method for setting up a trail note (see 'Organization of backtracking' earlier). The value of the **TrailData** argument will be stored in the trail note, and the same value will be passed back to the solver in the corresponding **backtrack** call.

Note that this service requires that the identity of the solver be known to the dispatcher (no **Sid** argument is passed).

```
void signal_numbered_error(long ErrNumber)
```

This is just a commodity function, an envelope for calling **xyp_signal** as

```
xyp_signal(xyp_clp_sys_error(xyp_mk_int(ErrNumber)));
```

for raising 'clp_system_error' with the integer number **ErrNumber** as 'other-info'. This function can be used for signaling internal errors with numerically coded error message (useful during development mainly).

NOTES:

1. By a process of gradual transformation, all regular services except **put_data_on_trail** have become 'self-describing'. For the sake of regularity of the interface, this last one also might be rendered self-describing, but the **sign_on/sign_off** protocol is still needed for **reset** (and maybe for later extensions).
2. It seems now that the function of **wake_instance** can be merged into the function of **sign_on**.

2.4 Support functions for arithmetic evaluation ("arithm.h")

The main advantage of using the system-provided arithmetic evaluation functions instead of implementing special evaluation for the solver is in consistency: the same operators and arithmetic functions are recognized and evaluated with the same error handling, range checking, etc.

It must be said, however, that the system functions are probably slower, so for simple calculations local evaluation is better.

Most of the support functions in this group return a `CspRetC` value as primary result; the actual result of arithmetic evaluation is placed into an output argument. The ‘normal’ outcome of the evaluation is indicated by `XXP_E_SUCCEED`, or, in the case of relation, also by `XXP_E_FAIL` (indicating that the relation does not hold, but the evaluation was performed normally). Any other return value indicates some error or exceptional condition encountered during evaluation. In this case the arithmetic module prepares an exception and returns the corresponding exception code (see chapter “The C Interface” in the User’s manual for details).

The caller can pass this exception code as return code to the system, where the exception will be raised immediately, can raise the signal directly, or can take corrective action itself (and reset the prepared exception), or prepare another exception to replace the original one.

There are two specific errors that a solver might want to handle directly. Let’s suppose that **Expr** is the evaluated term, and the return code is saved in the variable **rc**.

- (1) Unbound variable is encountered in **Expr**:

```
XXP_ERROR_KIND(rc) /* -> XXP_E_INSTANTIATION */
```

- (2) Constrained variable is encountered in **Expr**:

```
XXP_ERROR_KIND(rc) /* -> XXP_E_TYPE */
error_term = xxp_get_error_term(rc);
```

The prepared error term placed into **error_term** has the following form (in Prolog syntax):

```
type_error(evaluatable, CVAR)
```

where **CVAR** (the ‘culprit’) is the offending constrained variable (of type `XXP_T_RESTR`).

The evaluation of relation expressions requires that the main functor of the expression be passed as a separate argument in coded form. The symbolic code values are defined in the header file “`xxp_relc.h`”. The reason for this redundancy is that the solver usually has to analyze the expression anyway (and usually restricts the set of allowed relations), so it can have the code for no extra cost.

I. General term evaluation

```
CspRetC xxp_eval_expression(xxp_cell Expr, XxpNumType *Resultp)
```

Expr is a term; it is expected to be an arithmetically evaluable expression. If so, the function returns `XXP_E_SUCCEED`, and places the value of the expression into ***Resultp**. Otherwise the function prepares an exception and returns the corresponding exception code.

```
CspRetC xxp_eval_relation(unsigned RelCode, xxp_cell RelExpr)
```

RelExpr is a term; it is expected to be a structure the main functor of which is a (binary) arithmetic relation and the arguments of the structure should be arithmetically evaluable expressions. **RelCode** is the coded value corresponding to the main functor (the correspondence is not checked by the function).

If no error or exceptional condition is encountered during the evaluation, then the function returns either `XXP_E_SUCCEED` (when the relation holds between the arguments) or `XXP_E_FAIL` (when the relation does not hold). Otherwise the function prepares an exception and returns the corresponding exception code.

II. Evaluation of simple expressions with numeric arguments

```
CspRetC xxp_eval_unary_expr(xxp_cell func, XxpNumType narg,
                          XxpNumType *Resultp)
```

This function can be used to evaluate a simple unary expression with a numeric argument. **func** is the main functor of the expression, as e.g. (-)/1 or 'abs'/1; **narg** is the number constituting the argument of the expression.

The role of **Resultp** and the return code are the same as for **xxp_eval_expression** above.

```
CspRetC xxp_eval_binary_expr(xxp_cell func, XxpNumType narg1,
                             XxpNumType narg2, XxpNumType *Resultp)
```

This function is similar to **xxp_eval_unary_expr** above, but for binary expressions. **narg1** and **narg2** are the numbers constituting the first and the second arguments of the expression, respectively.

```
Boolean xxp_eval_binary_relation(unsigned RelCode,
                                XxpNumType Left, XxpNumType Right)
```

This function can be used to evaluate a simple relational expression, with numeric arguments. **RelCode** is the same as for **xxp_eval_relation** above; **Left** and **Right** are the first and the second arguments of the expression, respectively.

The function returns **TRUE** if the relation holds, otherwise the return value is **FALSE**

III. Helper functions

```
xxp_cell xxp_checked_mk_num(XxpNumType val,
                           unsigned argno, CspRetC *rcp)
```

This is a commodity function enveloping **xxp_checked_mk_int** and **xxp_mk_float** (see "The C Interface").

```
CspRetC xxp_get_checked_num_val(xxp_cell cell, unsigned argno,
                              XxpNumType *Resultp)
```

This is a commodity function enveloping **xxp_int_val** and **xxp_checked_float_val** (see "The C Interface"), with the roles of the return code and the output argument reversed (in order to avoid returning structure from the function).

IV. 'Quick' unification for numbers

```
CspRetC xxp_unify_num(XxpNumType val, xxp_cell xc, unsigned argno)
CspRetC xxp_unify_int(XxpNumType iv, xxp_cell xc, unsigned argno)
CspRetC xxp_unify_float(XxpNumType fv, xxp_cell xc)
```

These functions can be used instead of the general **xxp_unify** function when one of the terms to be unified is known to be a number, or more specifically an integer or a float number, respectively. The functions make use of the additional knowledge so they are more efficient. In some cases, float creation is avoided.

The last two ('specific') functions do not check the actual type of the number passed in the first argument; they rely on the caller. When an integer value is being unified (first argument), the value is checked for the proper range, and an integer overflow exception is returned if the check fails. The argument position in the prepared exception data is taken from the **argno** argument.

3. Programming conventions

In a later phase of the CS-Prolog development process, we became aware of the possibility of external name conflicts between the runtime core and extensions implemented by the user or other developers (e.g. implementation of foreign predicates). From that time we tried to restrict the form of external names used in the core, to specific forms, beginning with the prefix **Xxp**, **XXP**, **xpp_**, and the like, but there are still a lot of undisciplined external symbols around. In any case, it is highly recommended that the solver program also use some kind of common prefix, hopefully differing from any other extensions with which it might have to live together.

4. Macro definitions for compiling the sources

The following macros should be defined e.g. on the command line invoking the C compiler:

`-DRUNT -DCONSTRAINT`

For the compilation of the CLP configuration module “clp_cfg”, the definition of `CONSTRAINT` above sets a default value of `N_SOLVERS=1`. If this is inappropriate, then either an additional command line definition should be given, e.g.

`-DNSOLVERS=2`

or the definition of `N_SOLVER` in “clp_cfg.c” can be changed and activated (‘uncommented’).