DMCAI CLP 1.2

This Manual documents a variant of SICStus Prolog version 2.1,
featuring extensible unification

Edition 1.3
January 1994

by Christian Holzbaur

Department of Medical Cybernetics and Artificial Intelligence
University of Vienna
Freyung 6
A-1010 Vienna, Austria

1 Introduction

Within the version of SICStus Prolog *[Carlsson & Widen 90]* described in this manual, the unification mechanism has been changed in such a way that the user (you) may introduce interpreted terms and specify their unification through Prolog predicates. Extensible unification in turn, aims at the implementation of instances of the general CLP scheme *[Jaffar & Michaylov 87]*.

The basic mechanism that was utilized to provide extensible unification in the SICStus Prolog implementation are so-called *attributed variables*.

> If the reader is familiar with *meta-structures [Neumerkel 90]*, that likewise aim at the provision of extensible unification, s/he is referred to See Chapter 3 [Meta-structures], page 3, where we briefly compare the two approaches.

Attributed variables are variables with an associated attribute, which is a term. Attributes are attached to variables, and attributes are referred to, through built-in predicates. As far as the rest of a given Prolog implementation is concerned, attributed variables behave like variables — they can be considered as a subtype of type variable. The indexing mechanism treats variables and attributed variables the same way. Built-in predicates observe attributed variables as if they were ordinary variables. Special treatment for attributed variables applies in the following situations:

- During unification. This is the most important difference! When an attributed variable is to be unified with another attributed variable or some other non-variable term, user-defined predicates specify how this unification has to be performed.
- When printed via `print/1`, a user-supplied predicate gets a chance to print the attributed variable in some customized fashion.
- During memory management, i.e, during garbage collection, *early reset* and *variable shunting* are performed on attributed (and other) variables.

SICStus2.1 did not provide attributed variables directly, but within the kernel this data type was used to implement `freeze/2` and `dif/2` and other coroutining facilities. Although the data type *attributed variable* (variables with tag `CVA` for kernel hackers) could be utilized directly, the *wake mechanism* had to be changed for the new purpose. The wake mechanism is concerned with the treatment of attributed variables during unifications. The mechanism built into the original SICStus2.1 release is sufficient for the implementation of the mentioned coroutining mechanisms. For our CLP applications however, it had to be changed somewhat. If you are mainly interested in CLP programming, you don't want to know what exactly the changes are (you can always look at the code). An important consequence of the modifications is that the original coroutining mechanisms do not work in our SICStus clone. This is only half as bad as it sounds because explicit Prolog versions of comparable mechanisms, implemented via the new CLP scheme, will be provided. The **explicit** encoding of, say `freeze/2`, is not to be understood as a mere substitute for the original C code. The true motivation for explicit encodings is that it enables the user to combine, say `freeze/2`, with other instances of the general CLP scheme like CLP(R).

2 Manipulation of attributed variables

The following predicates provide for the introduction, detection, and manipulation of attributed variables.

`get_attribute(`X`,`C`)`

If X is an attributed variable, unify the corresponding attribute with C, otherwise the predicate fails.

`attach_attribute(`X`,`C`)`

Turn the free variable X into an attributed with attribute C. C must **not** be a variable. Note that attaching an attribute to variable changes the identity of the variable. Example:

```
| ?- write(X),nl,attach_attribute(X,prime),write(X),nl.
_40
_219
```

In terms of Logic, this change of identity does no harm as it takes place consistently. That is, all occurrences of the affected variable get the new identity. If some application depends on the ordering of terms, and variables in particular, attaching attributes to variables might invalidate the ordering of ordered data structures.

`detach_attribute(`X`)`

Remove the attribute from an attributed variable, turning it into a free variable. Note that detaching the attribute from a variable changes the identity of the variable. Example:

```
| ?- attach_attribute(X,prime),write(X),nl,
     detach_attribute(X),write(X),nl.
_328
_337
```

In terms of Logic, this change of identity does no harm as it takes place consistently. That is, all occurrences of the affected variable get the new identity. If some application depends on the ordering of terms, and variables in particular, detaching attributes from variables might invalidate the ordering of ordered data structures.

`update_attribute(`X`,`C`)`

Change the attribute of the attributed variable X to C. Acts as if defined below, but might be more (memory) efficient. C must **not** be a variable. Note that updating the attribute of an attributed variable changes the identity of the variable.

```
update_attribute( X, C) :-
  detach_attribute( X),
  attach_attribute( X, C).
```

3 Attributed variables vs. Meta-structures

In *[Holzbaur 90]* meta-structures on top of C-Prolog were applied for the specification and implementation of a variety of instances of the CLP scheme. The quality and the availability of SICStus Prolog motivated the migration to SICStus Prolog, entailing a compiler, garbage collector, and better arithmetics. In order to keep the implementation efforts within reasonable bounds, we decided to provide extensible unification on top of SICStus Prolog through the use of attributed variables as described above. Although attributed variables and meta-structures share some properties, there are differences.

> Meta-structures are ordinary, non-variable Prolog terms with the sole difference that they can be detected as members of this special sort. Meta-structures are introduced by a declaration `:- meta_functor N/A`, where `N/A` denotes any functor. The behavior of meta-structures during unifications can be specified precisely through a Prolog meta interpreter which makes unification explicit, and makes some further conventions integral parts of the specification *[Holzbaur 90]*.
>
> The implementations of many CLP instances require the functionality that is achieved with destructive updates in traditional, procedural realizations. In logic programming, this functionality is provided in a sound fashion by either copying or *modifying by variable substitution*. The latter option can be applied to meta-structures through the convention that one particular argument of the structure is a free variable, which will eventually be bound to another meta-structure, obeying the same convention. Therefore, sequences of modifications lead to meta-structure chains. The current 'value' of a meta-structure is to be found at the end of the chain. Traversing this chains could of course be left to the user, but it is so common a pattern, that is has been made part of the specification.

Please compare the implementation of `freeze/2` via meta-structures with the one via attributed variables:

```
:- meta_functor( frozen/2).

freeze( frozen(_,Goal), Goal).

meta_term_unify( frozen(Value,Goal), Value) :-
  call( Goal).

meta_meta_unify( frozen(V,G1), frozen(V,G2)) :-
  V = frozen(_,(G1,G2)).
```

The above encoding of `freeze/2` with meta-structures assumes the specification from *[Holzbaur 90]* being in force. Within the framework documented in this manual, `freeze/2` can be coded like this:

```
freeze( X, Goal) :-
  attach_attribute( V, frozen(V,Goal)),
  X = V.
```

```
verify_attribute( frozen(Var,Goal), Value) :-
  detach_attribute( Var),
  Var = Value,
  call(Goal).

combine_attributes( frozen(V1,G1), frozen(V2,G2)) :-
  detach_attribute( V1),
  detach_attribute( V2),
  V1 = V2,
  attach_attribute( V1, frozen(V1,(G1,G2))).
```

3.1 Source transformations

The most significant difference between meta-structures and attributed variables is that interpreted terms can be represented directly through meta-structures. When we realize interpreted terms via attributed variables, we have to replace interpreted terms through attributed variables with the interpreted terms as their attributes, say. For a given CLP instance with a given set of interpreted functors, one would typically use the `term_expansion/2` mechanism to perform this transformation. The same transformation applies to queries. In order to enable the user to toggle between a Prolog system which recognizes interpreted functors and one that does not, we provide an alternative toplevel, see Chapter 5 [Toplevel], page 7.

4 Unification in the presence of attributed variables

Once attributed variables have been created, they have to be dealt with during unification. The following procedural description lists the sequence of events from the point of view of the *WAM*.

- A unification between an unbound variable and an attributed variable binds the unbound variable to the attributed variable.
- When an *attributed variable* is about to be bound during unification because it is equated with a non-variable Prolog term or another attributed variable, the attributed variable and the value it should be bound to are recorded in some internal data structure.
- If there is more than one binding event for attributed variables between two inference steps, a list of attributed variable-value pairs is collected in some internal data structure.
- At the next inference step, the abstract machine takes measures to feed the attributed variable-value pairs to user-supplied predicates. The data structures for the representation of the list of variable-value pairs can be reclaimed at this point.

4.1 User-defined predicates

The following predicates have to be supplied by the user. They specify the behavior, i.e., the meaning of attributed variables during unification.

4.1.1 Specifying unification for interpreted terms

`verify_attribute(`*C*`,`*T*`)`
: This predicate is called when an attributed variable with attribute *C* is about to be unified with the non-variable term *T*.

`combine_attributes(`*C1*`,`*C2*`)`
: This predicate is called when two attributed variables with attributes *C1*,*C2* are about to be unified.

Note that the two predicates are are not called with the attributed variables involved, but with the corresponding attributes instead. The reasons are:

- There are simple applications which only refer to the attributes.
- If the application wants to refer to the attributed variables themselves, they can be made part the attribute term. The implementation of `freeze/2` (See Chapter 3 [Meta-structures], page 3) utilizes this technique. Note that this does not lead to cyclic structures, as the connection between an attributed variable and it's attribute is invisible to the pure parts of the Prolog implementation.

- If attributed variables were passed as arguments, the user's code would have to refer to the attributes through an extra call to `get_attribute/2`.
- As the/one attribute is the first argument to each of the two predicates, indexing applies. Note that attributed variables themselves look like variables to the indexing mechanism.

4.1.2 Printing

In order to gain control over the printing of attributed variables, the user may define the predicate:

`portray_attribute(`*A*`,`*V*`)`

The predicate should either print something based on *A* or *V*, or do nothing and fail. In the latter case, the default printer (`write/1`) will print the attributed variable like an unbound variable, e.g. '`_673`'. *A* is the attribute of the attributed variable *V*.

4.1.3 Dumping

Also related to the presentation of the results of computations is the user-supplied predicate:

`dump(`*Term*`,`*Copy*`,`*Constraints*`)`

`dump/3` is intended to do almost the same as `copy_term/2` (a SICStus builtin predicate), with the difference that besides a copy a **list** with some representation of the constraints associated with the variables in the copy is produced. *Copy* and *Constraints* must not contain any attributed variables. Speaking in global terms, this predicate should produce a *domain specific representation* of the constraints. Because *Copy* and *Constraints* do not contain attributed variables, these terms are **pure** Prolog objects and can dealt with correspondingly. You might for example want to assert or retract constraints or process them in some other way with **pure** Prolog programs that don't know about attributed variables.

The new toplevel supplied with this patch tries to call `dump/3` to present the answer constraints. See Chapter 5 [Toplevel], page 7. If it fails, some domain independent representation of the constraints, i.e. attributed variables, gets printed instead.

5 New toplevel

The new toplevel is needed because the original one does not know about the new uses of attributed variables. Example:

```
| ?- attach_attribute(X,prime).

prolog:prime
```

The printout produced by the original toplevel is pretty useless. The new one shows us the relation between X and its attribute:

```
prolog:clp_top.
[Clp] ?- attach_attribute(X,prime).

attach_attribute(X,prime)
```

If not customized further via `dump/3` (See Section 4.1.3 [Dumping], page 6.), this is nothing but the description for the reconstruction of the answer constraints. This is all we can expect in the general case. A suitable definition of `dump/3` will be able to produce the following 'theory' specific result:

```
prolog:clp_top.
[Clp] ?- attach_attribute(X,prime).

prime(X)
```

`prolog:clp_top`

Calling `prolog:clp_top/0` enters a toplevel that differs from the original one in the following ways:

- The prompt `[Clp] ?-` indicates that the new toplevel is executing. In fact, the prompt P can be set via `unset(prompt(_),set(prompt(`P`))`. See Section 6.1 [Flags], page 9.
- Answer substitutions are presented in the usual way. *Answer constraints* are extracted via `dump/3` and printed.

Please note that the new toplevel resides in the internal module `prolog` to make its very own implementation invisible to the debugger.

The new toplevel times the execution of goals. If these messages are bothering you, they can be switched off with the following definition:

```
user:portray_message(informational, timed(_)).
```

Just before entering the new toplevel, you might want to set the flag `expansion(clp)` (See Section 6.1 [Flags], page 9.). This flag enables `term_expansion/2` to 'know' whether the expansion has to deal with interpreted functors or not. On return from `prolog:clp_top/0`, the flag should be cleared again in order to return to the ordinary Prolog interpretation of terms.

The new toplevel catches exceptions. When left via `abort`, the flag indicating the expansion mode will not be cleared. In this hopefully rare cases, the flag has to be reset manually with `unset(expansion(_))`. Alternatively, you may use the coding scheme below, where the exception `reserved(3)` denotes an `Abort` event. Note that the exception code might be changed by Sicstus at any time.

```
top :-
  set( expansion(clp)),
  on_exception( reserved(3), prolog:clp_top, top_cleanup),
  unset( expansion(_)).

top_cleanup :-
  unset( expansion(_)),
  raise_exception( reserved(3)).  % proceed with exception handling
```

6 Other additions to SICStus2.1

Although not directly related to the general CLP scheme provided with this patch, some utility predicates have been added also.

6.1 Global flags

A simple mechanism for *global flags* is provided:

`set(`*Flag*`)`
: Sets *Flag* which can be any Prolog term.

`unset(`*Flag*`)`
: Clears all flags that unify with *Flag*.

`setting(`*Flag*`)`
: Nondeterministically unifies *Flag* with any flag set via `set/1`.

6.2 Numerical functions

`gcd(`*A*`,`*B*`)`
: The numerical function `gcd/2` has been added to the set of functions recognized by `is/2` and the predicates that perform numerical comparisons. The function computes the greatest common divisor of the two integer arguments *A* and *B*. Non-integer arguments rise a condition. Example:

```
| ?-  X is gcd( 2376446732, -23764).

X = 4
```

The motivation for the existence of this predicate is the absence of arithmetic functions and comparison predicates for rational numbers in SICStus2.1. Writing a rational evaluator in Prolog is easy, but efficiency is awful if `gcd/2` has to be coded in Prolog. Once SICStus reaches the numerical maturity of Commonlisp, patches like this one will be obsolete.

6.3 Type recognition predicate

`type(`*X*`,`*Type*`)`

The meta-logical predicate determines the type of *X* and unifies *Type* with one of the atoms `var`, `cva`, `float`, `integer`, `structure`, `atom`, `list`. `cva` means that *X* is an attributed variable. The remaining type names should be self explaining. The motivation for the existence of this predicate is that indexing in SICStus2.1 is not exhaustive on this list of types. With the help of this predicate we can write code like below, in order to get the benefits of indexing.

```
sample( X) :-
  type( X, Type),
  sample( Type, X).

  sample( var,     X) :- format( "~p is a variable", [X]).
  sample( atom,    X) :- format( "~p is an atom", [X]).
  sample( integer, X) :- format( "~p is an integer", [X]).
  sample( list,    X) :- format( "~p is a list", [X]).
```

References

[Carlsson & Widen 90]
Carlsson M., Widen J.: Sicstus Prolog Users Manual, Swedish Institute of Computer Science, SICS/R-88/88007C, 1990.

[Holzbaur 90]
Holzbaur C.: Specification of Constraint Based Inference Mechanisms through Extended Unification, Dept. of Medical Cybernetics & AI, University of Vienna, Dissertation, 1990.

[Holzbaur 92a]
Holzbaur C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification, in Bruynooghe M. & Wirsing M.(eds.), Programming Language Implementation and Logic Programming, Springer, LNCS 631, pp.260-268, 1992.

[Holzbaur 92b]
Holzbaur C.: A High-Level Approach to the Realization of CLP Languages, in Proceedings of the JICSLP92 Post-Conference Workshop on Constraint Logic Programming Systems, Washington D.C., 1992.

[Holzbaur 93a]
Holzbaur C.: Extensible Unification as Basis for the Implementation of CLP Languages, in Baader F., et al., Proceedings of the Sixth International Workshop on Unification, Boston University, MA, TR-93-004, pp.56-60, 1993.

[Holzbaur 93b]
Holzbaur C.: Using Constraint Logic Programs in Model-Based Reasoning, in Proceedings of the 4th Workshop on Artificial Intelligence and Knowledge-Based Systems for Space, European Space Agency, Simulation & Electrical Facilities Division, Noordwijk, ESA WPP-050, pp.239-247, 1993.

[Jaffar & Michaylov 87]
Jaffar J., Michaylov S.: Methodology and Implementation of a CLP System, in Lassez J.L.(ed.), Logic Programming - Proceedings of the 4th International Conference - Volume 1, MIT Press, Cambridge, MA, 1987.

[Neumerkel 90]
Neumerkel U.: Extensible Unification by Metastructures, Proc. META90, 1990.

Appendix A complete example

This example illustrates the realization of a very simple type system with the DMCAI CLP package. The types we want to deal with are `odd` and `even`. They partition the Prolog type `integer` into two disjoint sets. The syntax for associating a type with a term shall be `odd(X)` and `even(X)` respectively. Therefore, `odd/1` and `even/1` are our interpreted functors.

A.1 Sample session

```
SICStus 2.1 #8 [DMCAI Clp 2.1.3]: Mon Jan 31 11:15:19 MET 1994
| ?- use_module(types).
compiling types.pl...
compiling settings.pl...
compiled settings.pl in module settings, 83 msec 8208 bytes
types.pl compiled, 816 msec 15792 bytes

yes
| ?- top.
clp(enter)
[Clp] ?- f(odd(X),even(Y)) = Z.

Z = f(X,Y),
even(Y),
odd(X) ? ;

no
[Clp] ?- f(odd(X),even(Y)) = Z, Z = f(13,_).

X = 13,
Z = f(13,Y),
even(Y)
```

A.2 Necessary definitions

We start the example by providing the context for the new toplevel (See Chapter 5 [Toplevel], page 7).

```
%
% Simple type theory to demonstrate some CLP basics
%

:- use_module( settings).

top :-
  set( expansion(clp)),
    prolog:clp_top,
  unset( expansion(_)).
```

Next, we specify the source transformations that replace interpreted terms with attributed variables (See Section 3.1 [Source transformations], page 4.):

```
 1 term_expansion( A, B) :-
 2   setting( expansion( clp)),
 3   our_term_expansion( A, B).
 4
 5 our_term_expansion( X,       X) :- var( X), !.
 6 our_term_expansion( odd(X),  New) :- !,
 7   attach_attribute( New, odd(New)),
 8   New = X.
 9 our_term_expansion( even(X), New) :- !,
10   attach_attribute( New, even(New)),
11   New = X.
12 our_term_expansion( Old,     New) :-
13   functor( Old, N, A),
14   functor( New, N, A),
15   our_term_expansion( A, Old, New).
16
17 our_term_expansion( 0, _,   _) :- !.
18 our_term_expansion( N, Old, New) :-
19   arg( N, Old, Oa),
20   arg( N, New, Na),
21   our_term_expansion( Oa, Na),
22   N1 is N-1,
23   our_term_expansion( N1, Old, New).
```

In line 2 above, we test whether our interpreted functors should be interpreted indeed. The actual transformation takes place in lines 6 to 11. You might wonder why the code in lines 6–8 is not just:

```
 1 our_term_expansion( odd(X),  X) :- !,
 2   attach_attribute( X, odd(X)).
```

Well, this is because we want the term `odd(33)`, which is equivalent to `33` in our 'theory', to be translated correctly. Therefore we introduce a fresh attributed variable *New* (lines 7 and 10) and unify it with *X* (lines 8 and 11). Note that in our example the transformation of `odd(33)` yields `33` after the call to `verify_attribute/2` (See Section 4.1.1 [Unification], page 5), triggered by the unification in line 8, has verified the 'oddness' of 33. The rest of the transformation is straightforward.

Now we specify how the interpreted terms are to be unified with the pure Prolog data types (lines 1 to 10) and with each other (lines 12 to 17).

```
 1 verify_attribute( odd(X), Term) :-
 2   integer( Term),
 3   0 =\= Term mod 2,
 4   detach_attribute( X),
 5   X = Term.
 6 verify_attribute( even(X), Term) :-
 7   integer( Term),
 8   0 =:= Term mod 2,
 9   detach_attribute( X),
10   X = Term.
11
12 combine_attributes( even(X), even(Y)) :-
13   detach_attribute( X),
14   X = Y.
15 combine_attributes( odd(X),  odd(Y)) :-
16   detach_attribute( X),
17   X = Y.
```

The only Prolog data type compatible with our types are the integer numbers (lines 2 and 7). `Odd` and `even` have the usual semantics (lines 3 and 8). If an integer is compatible with the type of X, we detach the attribute and unify X which is a free variable with the integer *Term* (lines 5 and 10). All other attempted unifications between `odd` and `even` typed variables and the remaining Prolog data types fail as they are not covered by `verify_attribute/2`.

Unifications between typed variables are dealt with by `combine_attributes`. Both variables must be of the same type (lines 12 and 15). Besides verifying compatibility, we must not forget that the attributed variables are to be the same — because of this very unification. We detach the attribute of one of the two attributed variables involved (lines 13 and 16). The choice is arbitrary. Then the variables can be unified without rising another call to `combine_attributes/2` (lines 14 and 17).

With the definitions up to this point, we would get the following interaction:

```
[Clp] ?- f(odd(X),even(Y)) = Z.

Z = f(X,Y),
attach_attribute(Y,even(Y)),
attach_attribute(X,odd(X))

[Clp] ?- f(odd(X),even(Y)) = Z, Z = f(13,_).

X = 13,
Z = f(13,Y),
attach_attribute(Y,even(Y)) ?
```

In order to get the more appealing output that was shown in the session transcript (See Section A.1 [Sample session], page 12), `dump/3` has to be coded, see Section 4.1.3 [Dumping], page 6.

The following encoding of `dump/3` uses a binary tree as dictionary to represent the association between the (attributed) variables in the original *Term* and the variables in the *Copy* and the list

of *Constraints*. The code as presented gets fooled by cyclic structures. It is a trivial matter to make it cycle-proof with the very same dictionary that is used for the variables (left as an exercise).

Once the copy has been produced (line 2), the dictionary is traversed to collect the attributes associated with the variables in the copy via `collect_constraints/3` (lines 39–45), which is formulated as a DCG.

```
 1 dump( Term, Copy, Constraints) :-
 2   copy( Term, Copy, Dict),
 3   collect_constraints( Dict, Constraints, []).
 4
 5 copy( Term, Copy, Dict) :-
 6   type( Term, Tt),
 7   copy( Tt, Term, Copy, Dict).
 8
 9 copy( cva,       Cva,    Copy,     Dict) :-
10   get_attribute( Cva, Attrib),
11   copy_cva( Attrib, Copy, Dict).
12 copy( var,       V,      Copy,     Dict) :-
13   dict_insert( Dict, v(V,Copy), _).
14 copy( integer,   I,      I,        _).
15 copy( float,     F,      F,        _).
16 copy( atom,      A,      A,        _).
17 copy( list,      [X|Xt], [Xc|Xct], Dict) :-
18   type( X, Xtype),   copy( Xtype, X, Xc, Dict),
19   type( Xt, Xttype), copy( Xttype, Xt, Xct, Dict).
20 copy( structure, Term,   Copy,     Dict) :-
21   functor( Term, N, A),
22   functor( Copy, N, A),
23   copy_arg( A, Copy, Dict, Term).
24
25 copy_arg( 0, _,    _,    _) :- !.
26 copy_arg( N, Copy, Dict, Term) :-
27   N1 is N-1,
28   arg( N, Copy, Ac),
29   arg( N, Term, At),
30   type( At, Att),
31   copy( Att, At, Ac, Dict),
32   copy_arg( N1, Copy, Dict, Term).
33
34 copy_cva( odd(X), Copy, Dict) :-
35   dict_insert( Dict, cva(X,Copy,odd(Copy)), _).
36 copy_cva( even(X), Copy, Dict) :-
37   dict_insert( Dict, cva(X,Copy,even(Copy)), _).
38
39 collect_constraints( Dict)        --> var(Dict), !.
40 collect_constraints( t(L,Key,R) ) -->
41   (  Key = v(_,_)
42   ;  Key = cva(_,_,Pred) , [ Pred ]
43   ),
44   collect_constraints( L),
45   collect_constraints( R).
```

The dictionary operations on the binary tree can be performed with the following predicate:

```
% insert a F(Key,...) elem into the a binary tree
%
dict_insert( Tree, Key, Occ) :- var(Tree), !,
  Tree = t(_,Key,_),
  Occ = new.
dict_insert( t(L,Key0,R), Key, Occ) :-
  arg( 1, Key, Ak),
  arg( 1, Key0, Ak0),
  compare( Rel, Ak, Ak0),
  ( Rel = =, Occ = old, Key = Key0
  ; Rel = <, dict_insert( L, Key, Occ)
  ; Rel = >, dict_insert( R, Key, Occ)
  ).
```

What is still missing is how to print our attributed variables that stand for interpreted terms, see Section 4.1.2 [Printing], page 6. In general it is a good idea to use `dump/3` for that purpose. In **this** simple case however, it is not useful to apply the full dump machinery. Therefore:

```
portray_attribute( odd(X),  _) :- write( odd(X)).
portray_attribute( even(X), _) :- write( even(X)).
```

Note the use of `write/1`! If `print/1` is used instead, nothing useful will be printed (guess why).

Taken together, these code fragments implement our type 'theory'.

Warning:

- Because this code contains interpreted terms (`odd/1` and `even/1` functors occur in some predicates), it **must** be loaded from the original SICStus Prolog toplevel. Only after consulting or compiling the file, you would enter the new toplevel via `top/0`. If your programs behave strange, it is quite likely that a 'pure' program has been loaded into a context with interpreted terms, or vice versa.

Appendix B How to produce the DMCAI CLP clone

The following procedure assumes that you have got SICStus2.1 at patchlevel #3 or later. We will try to keep up with future versions of SICStus2.1.

1. Produce a copy of your original SICStus2.1 directory tree. Assuming that that the original resides in the directory '`sicstus2.1`', and that you want to call the clone '`sicstus2.1.clp`', you might use the commands:

   ```
   mkdir sicstus2.1.clp
   (cd sicstus2.1; tar cf - .) | (cd sicstus2.1.clp; tar xf -)
   ```

2. `cd sicstus2.1.clp`
3. execute the patch script

   ```
   patch < clp.2.1.8.patch
   ```

   The last digit in '`clp.2.1.8.patch`' identifies the SICStus2.1 patchlevel to which the patch applies. If you get messages about failed hunks (most likely for Makefiles), the corresponding patches must be applied by hand. You can check for failed patches with

   ```
   find . -name '*.rej' -print
   ```

4. `(cd Emulator; make realclean)`
5. `(cd Compiler; make realclean)`
6. `rm Library/*.ql Compiler/*.ql`
7. cd to the root of the copy (sicstus2.1.clp), ensure that the *OPTIONS* in the Makefile don't include `-DBDD`, and type:

   ```
   make all
   ```

   This step assumes that the executable of your original SICStus2.1 is installed as '`/usr/local/bin/prolog`'. If this is not so, add `PROLOG=someotherpath` to the make parameters.

   This step is where you could select native code generation. Check the Sicstus README for details.

Please let us know about how your installation went. Address correspondence to:

Christian Holzbaur
Dept. of Medical Cybernetics & Artificial Intelligence (DMCAI)
University of Vienna
Freyung 6
A-1010 Vienna
Austria
Email: christian@ai.univie.ac.at

Predicate Index

Concept Index

Table of Contents