A Conservative Approach to Meta-Programming in Constraint Logic Programming

Pierre Lim and Joachim Schimpf

European Computer-Industry Research Centre
Arabellastraße 17, 81925 München, Germany
{pierre,joachim}@ecrc.de

Abstract. Constraint Logic Programming [4] extends Logic Programming by generalizing the notion of unification to constraint solving. This is achieved by fixing the interpretation of some of the symbols in the language. The two alternative mechanisms used in the currently implemented CLP systems to achieve this operation are: (1) fix the interpretation before the program executes or (2) fix the interpretation at a point during program execution when it is used in a constraint. CLP($\mathcal{R}$) [5] and Prolog-III [1] take the first approach whereas CHIP [2] takes the second approach. The problem with the first approach is that interpreted terms cannot be manipulated syntactically. The problem with the second approach is that all constraint operations have to be made explicit and this increases the difficulty of programming. We propose a synthesis of both approaches that overcomes their individual difficulties. Our method is implemented in the ECL^{i}PS^e compiler system.

1 Introduction

The fundamental operation of unification in Logic Programming (LP) has been generalized to constraint solving in Constraint Logic Programming (CLP) [4]. Although this generalization greatly improves the efficiency and utility of CLP languages compared to LP languages it also complicates meta-programming. The problem is to decide how and when to assign the fixed interpretations of some of the functors. For example, the functors `1`, `2` and `+` in an arithmetic CLP language are interpreted respectively as the arithmetic constants one, two and the addition function. So the equation `1 + 2 = X + Y` is equivalent to `3 = X + Y`. However, for meta-programming the symbols `1`, `2` and `+` should be treated simply as uninterpreted symbols, so that the equation `1 + 2 = X + Y` has the solution { `X = 1, Y = 2` }. It is *not* equivalent to `3 = X + Y` which is unsatisfiable. The reconciliation of this overloading of functors is addressed by Heintze et al. [3] in which they give a theoretical framework for the problem and discuss a solution for the CLP($\mathcal{R}$) language. The problem with their method is that it is not conservative i.e. it does not preserve the current LP meta-programming functionality, but rather it defines new functionality to replace that which was lost. The conservation of current functionality is important because it means

that tools, techniques and applications developed for LP systems are usable on CLP systems. On the other hand, CHIP which distinguishes constraints syntactically has no problem with meta-programming but every constraint operation has to be made explicit, i.e all head unifications are syntactic not semantic. This is counter-intuitive if one expected, say, the + symbol to denote addition. Moreover the requirement for explicit constraint operations places an extra burden on the programmer.

We present a simple syntactic transformation which achieves a synthesis of both approaches and overcomes their individual difficulties and provide an implementation in the ECL^{i}PS^e[1] system. Our presentation is organized in the following way. First, we define the class of structures we are dealing with, i.e. those containing uninterpreted functors. The extensions to unification required by CLP are then discussed. Next, the approach of [3] is briefly reviewed. We use their theoretical basis in further discussions of the meta-programming problem and the solution. The CHIP approach is then discussed and be build on this approach to develop our solution. Our solution and its implementation in ECL^{i}PS^e is then given. In sections 7 and 8 we present a comparison with the approach of [3] and give our solutions to their examples. Finally some concluding remarks are made and a summary of our results is given.

2 Structures with uninterpreted functors

The fundamental extension of LP to CLP is the assignment of a non-Herbrand interpretation to some of the function symbols in the language and the inclusion of relations other than syntactic equality (according to a given algebraic description called the structure of computation). Of particular importance is the structure of the Herbrand Universe (HU) since this is the core of the Prolog programming language. In order to utilize Prolog programming techniques uninterpreted functors have to be included. We define the class of structures with uninterpreted functors which we denote parametrically as HU($\mathcal{D}$) where $\mathcal{D}$ represents the underlying algebraic structure e.g. rationals, reals, finite domains. Prolog has the structure HU($\perp$) since there is no structure under that of the uninterpreted functors.

We now give some definitions and then proceed to consider the types in these structures. A sort is a name of a type and a signature is a sequence of sorts. The alphabet of a CLP(HU($\mathcal{D}$)) language is partitioned into several classes.

- Π is the set of uninterpreted (programmed) predicate symbols, e.g. `laplace`, `fibonacci`, `nqueens`.
- $\Pi_{\mathcal{D}}$ ($\Pi_{\mathcal{D}} \cap \Pi = \emptyset$) is the set of interpreted predicate symbols and contains at least `=` (syntactic equality) in addition to any other predicates in $\mathcal{D}$, e.g. for the rational arithmetic structure in CHIP the following symbols denote

[1] ECL^{i}PS^{e}is the platform on which work on constraint handling is being performed at ECRC.

the usual arithmetic equality and inequality relations: { `$<=`, `$<`, `$>`, `$>=`, `$=` }.
- Σ is the set of uninterpreted function symbols e.g. `typeDevice`, `relay` and `[]`. Constant symbols are 0-ary function symbols.
- $\Sigma_{\mathcal{D}}$ ($\Sigma_{\mathcal{D}} \cap \Sigma = \emptyset$) is the set of interpreted function symbols, e.g. for the rationals they are {`+`,`-`,`*`,`/`} $\cup \mathcal{R}_C$ where $\mathcal{R}_C$ is the set of constant symbols for the rational numbers.
- V is the set of variable symbols. We adopt the Prolog convention that all identifiers beginning with an uppercase letter or an underscore are variable symbols.

The first issue is to decide the range of variables. For this we have to know what the types are. In HU($\mathcal{D}$) there are two types: $\mathcal{D}$ and $\mathcal{FT}$. $\mathcal{D}$ is the parametric type, e.g. for HU($\mathcal{R}$) $\mathcal{D}$ is the structure of the rational numbers. $\mathcal{D}$-terms are built from symbols in $\Sigma_{\mathcal{D}}$ (respecting signatures). $\mathcal{FT}$ is the type of finite trees over $\mathcal{D}$. $\mathcal{FT}$-terms are built from symbols in Σ and $\mathcal{D}$-terms, i.e. functors in Σ are constructors which can take as arguments either an (i) an uninterpreted constant, (ii) a $\mathcal{D}$-term or (iii) an $\mathcal{FT}$-term. The usual logical variables range over $\mathcal{FT}$-terms. We introduce a new kind of variable called a *solver-variable* which ranges over $\mathcal{D}$-terms. Note that solver-variables are atomic within $\mathcal{FT}$-terms.

3 Extended unification

For the class of structures HU($\mathcal{D}$) we have to distinguish when to send equalities resulting from head unification to the constraint solver for $\mathcal{D}$. This extension is summarized in the table below: *sv* abbreviates solver-variable and *unify* denotes the standard syntactic unification operation. As one would expect the essential operations are: D-terms are sent to the constraint solver for $\mathcal{D}$, unifications between $\mathcal{D}$-terms and $\mathcal{FT}$-terms fail, both $\mathcal{D}$-terms and $\mathcal{FT}$-terms are bound to variables and an equality between a solver-variable and an $\mathcal{FT}$-term fails.

Extended Unification Table				
`=$=`	$\Sigma_{\mathcal{D}}$	Σ	variable	solver-variable
$\Sigma_{\mathcal{D}}$	send to solver	**fail**	bind	send to solver
Σ	**fail**	*unify*	bind	**fail**
variable	bind	bind	bind	bind $v \rightarrow sv$
solver-variable	send to solver	**fail**	bind $v \rightarrow sv$	send to solver

4 The approach of Heintze et al.

The approach of [3] is to extend the underlying structure of computation for meta-programming and this is accomplished as follows.

- For every interpreted function symbol a new uninterpreted function symbol (called the $\mathcal{M}$-coded form) is added into Σ. For example, for $\Sigma_{\mathcal{D}} = \{$ `+`, `-`, `*`, `/` $\}$ we add the corresponding $\mathcal{M}$-coded forms $\{$ $\hat{+}$, $\hat{-}$, $\hat{*}$, $\hat{/}$ $\}$ to Σ. We shall follow the convention of [3] and denote $\mathcal{M}$-coded forms by placing a hat over the symbol.
- The function *quote* maps an interpreted function symbol to its $\mathcal{M}$-coded form.
- The function *eval* maps an $\mathcal{M}$-coded form back to its interpreted symbol.

The formal definitions (given by Heintze et al.) of *quote* and *eval* are given below. The $\mathcal{M}$-coded forms, *quote* and *eval*, and the axiom system below define a scheme of meta-programming structures of computation called $\mathcal{M}$ which can be added to any CLP language. An instance CLP($\mathcal{R} + \mathcal{M}$) is given by [3].

$$quote(t) = \begin{cases} V & \text{if t is the variable V} \\ \hat{f}(quote(t_1), \ldots, quote(t_n)) & \text{if } t \text{ is } f(t_1, \ldots, t_n), \\ & n \geq 0 \text{ and } f \text{ is interpreted} \\ f(t_1, \ldots, t_n) & \text{if } t \text{ is } f(t_1, \ldots, t_n), \\ & n \geq 0 \text{ and } f \text{ is uninterpreted} \end{cases}$$

$eval(\hat{f}(t_1, \ldots, t_n)) = f(eval(t_1), \ldots, eval(t_n)), n \geq 0$
$eval(g(t_1, \ldots, t_n)) = g(eval(t_1), \ldots, eval(t_n)), n \geq 0$
$eval(\hat{\text{quote}}(t)) = t$

Using the meta-programming structure above we say that CLP($\mathcal{R}$) is an eval-quote language, i.e. all symbols are interpreted unless explicitly quoted. Thus to facilitate meta-programming CLP($\mathcal{R}+\mathcal{M}$) provides the functions `quote` and `eval` and the following functionality (tabulated below).

Modified Functionality	
`nonground/1`	Fails if its argument has a unique value
`nonvar/1`	Succeeds if its argument is constrained
`var/1`	Fails if its argument is constrained
`rule/2`	Like `clause/2` of Prolog and produces $\mathcal{FT}$-terms
`assert/1`	Asserts the rule with the projection of the variables of the rule (from constraint store) conjoined in the body
`retract/1`	Retracts the rule using extended unification

New Functionality	
`coded_ccs/1`	Produces an $\mathcal{M}$-coded term representing the constraint store
`ground/1`	Succeeds if its argument has a unique value
`quoted_rule/2`	Like `rule/2` but produces $\mathcal{M}$-coded terms
`constructed/1`	Succeeds if its argument is bound to a structure
`unconstructed/1`	Fails if its argument is bound to a structure
`arithmetic/1`	Succeeds if its argument is a $\mathcal{R}$-term
`syntactic/1`	Fails if its argument is a $\mathcal{R}$-term
`quoted_retract/1`	Like retract but uses syntactic unification only

5 The CHIP approach

CHIP [2] is a quote-eval language, i.e. all symbols are quoted unless explicitly `eval`uated. However that there is no `quote` or `eval` function but instead the interpreted predicates (denoted by symbols in Π) `eval`uate their arguments. Note that the `eval` operation also marks (operationally the tag is changed) all variables as solver-variables[2]. For example, for the CHIP constraint `X + Y $= 6 * Z` involving the the rational arithmetic relation `$=/2` the following steps are performed.

1. Both arguments are `eval`uated, i.e. `eval(X + Y)` and `eval(6 * Z)`. This has the effect that the variables `X`, `Y` and `Z` are marked as solver-variables and the binary functors `+` and `*` get assigned their arithmetic interpretation.
2. The `eval`uated equality constraint is then added to the constraint store (i.e. the set of collected constraints) and a satisfiability check is made.

Since all symbols are quoted, there is no problem with meta-programming. However, this means that CHIP does not do semantic head unification at all, unlike CLP($\mathcal{R}$). However, all semantic head unifications can be shifted into the body where the interpreted predicates will `eval`uate correctly (see section 6.1 for the transformation). For example, the transformation of a program to compute Fibonacci numbers is given below where in CHIP the symbol `$>=` denotes the rational arithmetic relation for greater-than-or-equal-to.

The CLP($\mathcal{R}$) Fibonacci Program

```
fib(0,1).
fib(1,1).
fib(N,X1+X2) :-

   N >= 2,
   fib(N-1,X1),
   fib(N-2,X2).
```

The CHIP Fibonacci Program

```
fib(X,Y) :- X $= 0, Y $= 1.
fib(X,Y) :- X $= 1, Y $= 1.
fib(N,Y) :-
   Y $= X1 + X2,
   N $>= 2,
   fib(N-1,X1),
   fib(N-2,X2).
```

6 Our method and its implementation in the ECL^{i}PS^e compiler system

Since not all clauses in a CLP program will use extended unification we make a distinction between those that have purely syntactic head unification, which we shall refer to as *ordinary clauses*, and those that use extended head unification, which we shall refer to as *constraint clauses*. In this way, we get the advantages of the eval-quote approach but with ordinary clauses we also get the usual LP term handling capability. In CLP($\mathcal{R}$) all clauses are constraint clauses. We distinguish constraint clauses in our language by a different neck operator `<-`. (See section

[2] This operation is trailed and undone on backtracking

8.2 for an example containing both kinds of clauses). Thus we can write the Fibonacci program as follows.

```
fib(0,1) <- true.
fib(1,1) <- true.
fib(N,X1+X2) <-
    N $>= 2,
    fib(N-1,X1),
    fib(N-2,X2).
```

The ECL^{i}PS^e system contains CHIP constraint handling functionality and is the platform currently used at ECRC to investigate constraint handling. Constraint clauses are handled by preprocessing with the objective of moving all extended unifications into the body. This is accomplished by using the global macro facility of the ECL^{i}PS^e compiler to expand all clauses with the `<-` neck (See Appendix A). However, a naive search for interpreted functors and replacement with a new variable produces incorrect results. Consider the following example. Since there are no interpreted functors in the head no preprocessing is done at all. But the query `max(1+3,1+1,2+2)` incorrectly fails against the transformed program because a syntactic unification is performed where a semantic unification should have been done.

Original CLP($\mathcal{R}$) Program	Transformed ECL^{i}PS^e Program

`max(X,Y,X) :-`
`  X >= Y.`
`max(X,Y,Y).`
	`max(X,Y,X) :-`
	`  X $>= Y.`
	`max(X,Y,Y) :-`
	`  true.`

We now formally present our transformation and argue that it is correct.

6.1 Transformation of constraint clauses

We split the description of our transformation into two cases. One where there is sufficient ground information to determine the type of unification and the other where there is not.

Case 1: The head argument is not a variable. If a subterm in the head of a constraint rule contains an interpreted symbol i.e. either an interpreted constant or an interpreted functor then we replace the term by a new variable and insert a solver call in the body. (See the example for the Fibonacci program above).

Case 2: The head argument is a variable. Here we consider the problem of aliasing. There are two cases, (i) where the call performs the alias and (ii) where the definition performs the alias. The first case occurs when the variable appears only once in the head e.g. `p(X,Y)`. In this situation we can simply perform

a binding since the only "atomic" object that exists is the multiply-occurring $\mathcal{D}$-term in the call i.e. there are no unifications between any head variables. So clauses such as:

```
p(X,Y) <- true.
```

simply have `<-` replaced by `:-`.

Case (ii) arises where a variable appears more than once in the head of a constraint clause. In this case there could be a unification between two head variables. (See the example for `max/3` above). Here we must move the extended unifications between head variables into the body, i.e. the decision for a syntactic or semantic unification is taken at runtime. For `max/3` the transformed program is given below.

```
max(X1,Y,X2) :-
    X1 =$= X2,
    X1 $>= Y.
max(X,Y1,Y2) :-
    Y1 =$= Y2,
    true.
```

7 Comparison with existing work

CLP($\mathcal{R}+\mathcal{M}$) modifies the functionality of a number of standard Prolog builtin predicates. The changes essentially involve extensions to the operations to cover the cases where constraints are involved. Compared with our approach we do not modify the builtins but instead can write new versions requiring the addition of a few new builtin predicates. We shall go through the list of builtin predicates that are modified in CLP($\mathcal{R}+\mathcal{M}$) (the list is in section 4). If we assume that variables are instantiated if they have a unique value then there is no need to change `nonground/1`. For `modified_var/1` we provide the following code.

```
modified_var(X) :-
    var(X),
    !.
modified_var(X) :-
    solver_variable(X).
```

In a similar way to `modified_nonvar/1` we provide `modified_var/1`.

```
modified_nonvar(X) :-
    not(modified_var(X)).
```

The code for `modified_rule/2`, `modified_assert/1` and `modified_retract/1` are given in Appendix B.

Some of the new functionality provided in CLP($\mathcal{R}+\mathcal{M}$) is redundant in ECL^{i}PS^e. `ground(X)` can be written as `not(nonground(X))` where `not/1` is

negation-as-failure. `quoted_rule/2` is the same as `clause/2`. `constructed(X)` can be written as `compound(X)` and `unconstructed(X)` can be written as `not(constructed(X))`. `syntactic/1` is a simple term inspection predicate much like `numbervars/3` which checks a term to make sure there are no (syntactic versions of the) interpreted functors in `X` and there are no solver-variables. (For solver-variables we need a new builtin, say `solver_variable/1`, which simply checks the tag). We have no need for explicit $\mathcal{M}$-coded forms in $\mathcal{FT}$-terms since all $\mathcal{D}$-terms are atomic. `arithmetic(X)` is written as `not(syntactic(X))`. `quoted_retract/1` is the same as `retract/1`.
Our approach has several advantages over that of [3].

1. Since we are conservative of the standard meta-programming functionality of Prolog, standard Prolog code will run without problems.
2. We do not modify the standard semantics of any of the Prolog builtin predicates. Again, this has the advantage of the point above.
3. Our approach is more flexible because instead of hard-coding new functionality into existing builtins we can write the required builtins *at the user level* on top of existing functionality.
4. Our approach conserves all existing Prolog optimizations including indexing. Since the transformation moves all semantic unifications into the body what is left in the head must be purely syntactic and so standard indexing techniques can be used to *discriminate between constraint clauses.*
5. Ordinary clauses not using constraints do not pay any performance penalty.
6. Our scheme has the advantage that all syntactic unifications are scheduled first. Since calling the constraint solver for $\mathcal{D}$ is usually more expensive than syntactic unification, if the unification fails due to some Herbrand constraint being violated then the $\mathcal{D}$ constraint solver will not be called.
7. Our approach offers the user the possibility of tailoring the constraint handling mechanism since the transformation can be performed manually to achieve any degree of mixed syntactic and semantic head unification handling.
8. Thus we do not need explicit $\mathcal{M}$-coded forms nor any explicit `quote` or `eval` function.

8 Examples

In this section we examine several example that have been given by [3] and discuss their implementation in ECL^{i}PS^e.

8.1 The standard meta-circular interpreter

We now compare the standard meta-interpreters as implemented for both CLP($\mathcal{R}+\mathcal{M}$) and ECL^{i}PS^e.

CLP($\mathcal{R}+\mathcal{M}$) meta-interpreter

```
goal(true).
goal((A,B)) :-
    goal(A),
    goal(B).
goal(X) :-
    constraint(X).
goal(X) :-
    rule(X,Z),
    goal(Z).

constraint(A = B) :-
    A = B.
constraint(A > B) :-
    A > B.
```

ECL^{i}PS^e meta-interpreter

```
goal(true).
goal((A,B)) :-
    goal(A),
    goal(B).
goal(X) :-
    constraint(X).
goal(X) :-
    clause(X,Z),
    goal(Z).

constraint(A = B) :-
    A = B.
constraint(A $= B) :-
    A $= B.
constraint(A $> B) :-
    A $> B.
```

The most noticeable differences are:

- We do not require a special version `rule/2` of `clause/2` in the fourth clause of `goal/1`. This is because the semantic unifications have been moved into the body where calls to the constraint solver for $\mathcal{D}$ can be treated like builtins.
- We add to `constraint/1` a clause for semantic equality, i.e. `$=`.
- We use the ECL^{i}PS^e symbols (e.g. `$>`) for rational constraints in the definition of `constraint/1`. The usual inequality symbols are already utilized by standard Prolog arithmetics.

8.2 Symbolic differentiation

Heintze et al. [3] give a program in CLP($\mathcal{R}+\mathcal{M}$) for the symbolic differentiation of a function in one variable as follows.

```
diff(T,0) :-
    ground(T).
diff(X,1) :-
    unconstructed(X),
    !.
diff(quote(A + B), quote(DADX + DBDX)) :-
    diff(A, DADX),
```

```
    diff(B, DBDX).
diff(quote(A * B), quote(DADX * B + DBDX * A)) :-
    diff(A, DADX),
    diff(B, DBDX).

?- Y = quote(X*X + 2*X + 1),
   diff(Y, DYDX),
   eval(DYDX) = 0,
   T = eval(Y),
   printf("Turning point: X = %, Y = % \n",[X,T]).
```

In ECL^{i}PS^e the program is as follows.

```
diff(T,0) <-
    ground(T).
diff(X,1) <-
    unconstructed(X),
    !.
diff(A+B,DADX+DBDX) :-
    diff(A,DADX),
    diff(B,DBDX).
diff(A*B,DADX*B+DBDX*A) :-
    diff(A,DADX),
    diff(B,DBDX).

?- Y = X*X + 2*X + 1,
   diff(Y,DYDX),
   DYDX $= 0,
   T $= Y,
   printf("Turning point: X = %d, Y = %d \n",[X,T]).
```

The code for `ground/1` and `unconstructed/1` are as given earlier. Note that in the third and fourth clauses there is no need in ECL^{i}PS^e to quote the arguments in the head. We simply write them as ordinary clauses; this has the effect of `quot`ing all head arguments. Since `=` means Herbrand unification there is no need to `quote` the second argument of the first goal in the query. There is no need to `eval` arguments in the third and fourth goals in the query because the $\mathcal{D}$ constraint solver automatically `eval`uates its arguments.

8.3 Partial evaluation

A technique of partial evaluation is also described by [3]. What is done is to execute a query and then use the simplified form of the answer to construct new rules. These new rules, of course, represent specializations w.r.t. the query. They give the following example.

```
resistor(V,I,R) :-
    V = I*R.

?- resistor(V,I1,R1), resistor(V,I2,R2),
   I = I1+I2,
   assert(parallel_resistors(V,I,R1,R2)).
```

This results in the following being asserted.

```
parallel_resistors(V,I,R1,R2) :-
    I = V/R1 + V/R2.
```

In ECL^{i}PS^e one can implement the above as follows.

```
resistor(V,I,R) <-
    V $= I*R.

?- resistor(V,I1,R1), resistor(V,I2,R2),
   I $= I1+I2,
   modified_assert(parallel_resistors(V,I,R1,R2) :- true).
```

What `modified_assert/1` does is to get the variables in the head, perform a projection of the constraint store w.r.t. these variables, append these constraints into the body creating a new body `B1` and then finally add this new *syntactic* clause into the dynamic database. Incidentally, it should be noted that `modified_assert/1` corresponds to the CLP($\mathcal{R}+\mathcal{M}$) `assert/1` extended to deal with constraints. The important point to note here is that instead of hard-coding the meta-programming functionality we provide two new builtins `solver_variable/1` and `projection/2`[3]. *Together with existing functionality* this is sufficient to program whatever specialized versions of the existing builtins is required *at the user level*. This makes our approach more flexible and allows better tailoring (of builtins) to application specifications.

9 Conclusion

The incorporation of constraint handling into logic programming systems is an important development and we have shown how it can be be integrated with existing technology such that the existing functionality is preserved and the standard environment, i.e. builtin predicates, is unchanged. We have discussed the two possible approaches to implementing the mechanism that assigns the fixed interpretation i.e. quote-eval and eval-quote. Each of the alternatives has drawbacks but by combining both approaches through context we have a synthesis which overcomes the problems of the individual approaches. We have

[3] `projection(V,C)` binds `C` to a copy of the constraint store in which all variables not in the list `V` have been eliminated. This domain-specific builtin is provided by the user.

presented an algorithm for implementing our method through a simple syntactic transformation, argued its correctness and given an implementation in ECL^{i}PS^e. Comparisons with existing CLP($\mathcal{R}+\mathcal{M}$) meta-programming examples have been made. In summary, the advantages of our approach are as follows.

- Standard Prolog code runs unchanged.
- We do not require explicit `quote` or `eval` functions.
- We do not require a complex meta-programming constraint solver to deal with equations explicitly containing calls to `quote` or `eval` as in CLP($\mathcal{R}+\mathcal{M}$).
- Our approach is more flexible (the user can perform the transformation manually to tailor the system to application specifications).
- We do not alter the standard environment.
- Code not using constraints does not pay a penalty.
- We schedule (usually cheaper) Herbrand unifications first, thereby short-circuiting calls to the $\mathcal{D}$-constraint solver in case of failure.
- We do not hard-code the meta-programming facilities into the system e.g. `modified_assert`.

Acknowledgements

We thank Alex Herold, Mark Wallace, Mireille Ducassé and Pascal Brisset for discussions and comments. This work was partially supported by Esprit Project 5291 CHIC.

References

1. Alain Colmerauer, "Opening the Prolog-III Universe", BYTE Magazine, August, 1987.
2. Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf and Françoise Berthier, "The Constraint Logic Programming Language CHIP", Proceedings of the 1988 International Conference on Fifth Generation Computer Systems, ICOT, 1988.
3. Nevin Heintze, Spiro Michaylov, Peter Stuckey and Roland Yap, "On Meta-Programming in CLP($\mathcal{R}$)", Proceedings of the 1989 North American Conference on Logic Programming, Cleveland, Ohio, USA, October 16–20, 1989.
4. Joxan Jaffar and Jean-Louis Lassez, "Constraint Logic Programming", Proceedings of the 1987 ACM Symposium on Principles of Programming Languages, Munich, January 1987, pp. 111–119.
5. Joxan Jaffar and Spiro Michaylov, "Methodology and Implementation of a CLP System", Proceedings of the 4$^{\text{th}}$ International Conference on Logic Programming, Melbourne, 1987, pp. 196–218.

Appendix A.

```
%
% Operator declaration for constraint clauses
%

:- op(1200,xfx,<-).

%
% Operator declarations for Rationals
%

:- op(700,xfy,$=).
:- op(700,xfx,$<=).
:- op(700,xfx,$<).
:- op(700,xfx,$>).
:- op(700,xfx,$>=).
:- op(700,xfy,=$=).

%
% Interpreted functors for the Rationals
%

interpreted(X) :- integer(X).
interpreted(_ + _).
interpreted(_ - _).
interpreted(_ * _).
interpreted(_ / _).

transform((Head <- Body), (NewHead :- NewBody)) :-
    functor(Head, F, A),
    functor(NewHead, F, A),
    find_semantic_unifs(A, Head, NewHead, [], SemUnifs, [],
                                                Varmap),
    find_aliases(Varmap, Aliases),
    add_goals(Aliases, Body, Body1),
    add_goals(SemUnifs, Body1, NewBody).

find_semantic_unifs(0, _, _, Goals, Goals, Aliases, Aliases) :-
    !.
find_semantic_unifs(N, Head, NewHead, Goals0, Goals,
                                      Aliases0, Aliases) :-
    arg(N, Head, Arg),
    arg(N, NewHead, Aux),
    find_semantic_unifs1(Arg, Aux, Goals0, Goals1,
                                   Aliases0, Aliases1),
```

```
    N1 is N-1,
    find_semantic_unifs(N1, Head, NewHead, Goals1, Goals,
                                           Aliases1, Aliases).

find_semantic_unifs1(X, Y, Goals, Goals, Al, [X=Y|Al]) :-
    var(X),
    !.
find_semantic_unifs1(X, Y, Goals, [Y $= X | Goals], Al, Al) :-
    interpreted(X), !.
find_semantic_unifs1(X, X, Goals, Goals, Al, Al) :-
    atomic(X), !.
find_semantic_unifs1(X, Y, Goals0, Goals, Al0, Al) :-
    functor(X, F, A),
    functor(Y, F, A),
    find_semantic_unifs(A, X, Y, Goals0, Goals, Al0, Al).

find_aliases([], []) :- !.
find_aliases(Aliases0, Aliases) :-
    sort(Aliases0, [First|More]),
    find_aliases(First, More, [], Aliases).

find_aliases(X=X, [], Al, Al) :- !.
find_aliases(X=Y, [X1=Y1|More], Al0, Al) :-
    ( X == X1 ->
        Al1 = [Y =$= Y1|Al0]
    ;
        X=Y,
        Al1 = Al0
    ),
    find_aliases(X1=Y1, More, Al1, Al).

add_goals([], Body, Body) :- !.
add_goals([Goal|Goals], Body0, (Goal , Body)) :-
    add_goals(Goals, Body0, Body).

:- define_global_macro((<-)/2,transform/2,[clause]).
```

The =$= predicate is a user-level predicate that reifies extended unification. It is only used in the case where there is a head unification between two variables. In this case, both variables could be $\mathcal{FT}$-terms requiring a combination of syntactic and semantic unification.

Appendix B.

```
interpreted(_ $= _).
interpreted(_ $< _).
interpreted(_ $> _).
interpreted(_ $<= _).
interpreted(_ $>= _).
interpreted(_ =$= _).

modified_clause(H,B) :-
    functor(H,F,A),
    find_semantic_unifs(A,H,NewHead,[],SemUnifs,[],VarMap),
    clause(NewHead,B),
    extended_unifs(SemUnifs),
    find_aliases(VarMap,Aliases),
    extended_unifs(Aliases).

extended_unifs([]) :- !.
extended_unifs([H|T]) :-
    call(H),
    extended_unifs([T]).

get_vars(T,VarsSeen,NewVars) :-
    modified_var(T),
    !,
    NewVars = [T|VarsSeen].
get_vars(T,VarsSeen,NewVars) :-
    functor(T,_,Arity),
    Arity > 0,
    !,
    get_vars_aux(Arity,T,VarsSeen,NewVars).
get_vars(T,Vars,Vars).

get_vars_aux(0,_,Vars,Vars) :- !.
get_vars_aux(N,T,Vars0,Vars) :-
    arg(N,T,Arg),
    get_vars(Arg,Vars0,Vars1),
    N1 is N - 1,
    get_vars_aux(N1,T,Vars1,Vars).

get_vars(T,V) :-
    get_vars(T,[],V1),
    sort(0,<,V1,V).

modified_assert(H :- B) :-
    get_vars(H,V),
```

```
    projection(V,C),
    add_goals(C,B,B1),
    assert(H :- B1).

goals_only((G1,G2),G,O) :-
    goals_only(G1,G,O1),
    goals_only(G2,O1,O).
goals_only(G,I,I) :-
    interpreted(G),
    !.
goals_only(G,I,[G|I]).

modified_retract(H :- B) :-
    functor(H,F,A),
    find_semantic_unifs(A,H,NewHead,[],SemUnifs,[],VarMap),
    clause(NewHead,Y),
    goals_only(B,[],BOnly),
    goals_only(Y,[],YOnly),
    B = Y,
    extended_unifs(SemUnifs),
    find_aliases(VarMap,Aliases),
    extended_unifs(Aliases),
    retract(NewHead :- Y).
```

This article was processed using the LaTeX macro package with LLNCS style