Attributed Equivalence Classes in Prolog Implementations of CLP Languages

Christian Holzbaur
Austrian Research Institute for Artificial Intelligence, and
Department of Medical Cybernetics and Artificial Intelligence
University of Vienna
Freyung 6, A-1010 Vienna, Austria
email: christian@ai.univie.ac.at

TR-92-27

Abstract

We introduce attributed equivalence classes as an explicit abstract data type for the representation and manipulation of general equation systems where the decision algorithm is based on quantifier elimination. The partition of quantifiers into equivalence classes results very naturally from a simple abstraction that separates the equation solving process in the object domain from global manipulation of equation systems. We propose and report on an implementation of a linear complexity equivalence relation maintenance algorithm within the framework of logic programming, based on extensible unification. The explicit representation of global aspects of equation systems leads to an object oriented approach to equation solving.

1 Introduction

Constraint Logic Programming (CLP) languages extend unification by a more general operation called constraint satisfaction [Cohen 90]. Theoretical foundations of CLP languages are provided by [Jaffar et al. 86] and [Colmerauer 90]. The algebraic structures whose introduction and treatment constitutes the extension of a given unification-based language governs the selection of the decision procedures used by the constraint satisfaction mechanism. In the case of Real Closed Fields and Booleans the decision procedures rest upon quantifier elimination [Boole 47, Tarski 48]. In the CLP context we deal with *systems* of equations over the algebraic domain expressions. A system of equations is a conjunction of

equations that share quantifiers, i.e. variables[1] in LP terminology. The solution of a system of equations consists of consecutive quantifier elimination steps that work on a single equation and express the eliminated quantifier in terms of the remaining quantifiers. This process partitions the set of all quantifiers into two disjoint sets of dependent and independent quantifiers. When the definition of the dependent quantifiers is substituted into the definition of the other dependent quantifiers we speak of a *solved form* representation of the system of equations. The solved form of a system is equivalent to the original system with regard to satisfiability and the set of solutions in general — In fact it is a compact representation of the solution space that facilitates the detection of satisfiability and the construction of solutions.

In the following sections we direct our attention towards the back substitution step that maintains the invariant about the solved form[2]. In particular, a programming technique that addresses the basic computational demand and the realization of back substitution in the context of logic programming, is introduced. In global terms this investigation is motivated by our attempt to realize CLP instances through extensible unification within (C)LP languages. The provision of sound mechanisms that produce the functionality that is achieved through destructive updates in procedural languages, is of prime importance.

In order to concretize the discussion to follow, we chose to describe solved form maintenance in CLP($\Re$) as a specific instance of the general CLP scheme. CLP($\Re$) is supposed to operate on the $\Re$eal numbers. Existing implementations are complete for linear (in)equations only. This restriction is motivated by the doubly exponential complexity of the decision algorithm for the general case. Nonlinear equations are delayed until they get linear, or sufficiently simple to be solved at last [Jaffar et al. 91].

2 Maintaining the solved form

Traditional procedural implementations of numerical linear equation solvers operate on the coefficient matrix of an equation system of given dimension. Gaussian (quantifier) elimination aims at the determination of the rank of the system by transforming the matrix into triangular from, which in turn, together with basic results from linear algebra, constitutes a decision algorithm for linear equations over real numbers. Note however, that the matrix of a linear system, transformed into triangular form, is not the solved form of the system as described above because the back substitution step is not performed by the basic triangularization. If it is made part of the elimination, we speak of Gauss-Jordan elimination. The triangular part above the diagonal labelled U in Figure 1 will be 0 after back substitution.

Within a CLP implementation, quantifier elimination has to be carried out in an incremental fashion as the constraints, i.e. the equations, are handled to the solver one by one in the

[1] We only deal with connected systems here as disconnected systems can be solved in isolation

[2] The invariant is that dependent quantifiers are expressed in terms of independent quantifiers

Figure 1: Gaussian and Gauss-Jordan elimination

course of a CLP program execution. A further deviation from the basic, classical scheme is that we typically deal with sparse systems where individual equations do not refer to all or almost all quantifiers of the system. This observations are reflected in the implementations of incremental versions of the elimination procedures and the data structures for the coefficient matrix.

In particular, for our implementation of CLP($\Re$) we use *ordered* lists of variable-coefficient pairs to represent the rows of the coefficient matrix. This data structure is optimal in the sense that it allows for multiplications of rows with scalars and additions of rows that are of complexity $O(n)$, n being the number of nonzero entries of the rows. Having abandoned the notion of a global matrix, we are left with the problem of how to perform back substitutions. The next two sections describe two principal means to perform back substitution in sparse matrices. The former is the scheme employed by [Heintze et al. 91] which is more suitable for procedural implementations because it involves structure mutation and explicit memory management. The latter, novel approach, has been designed with implementations of CLP languages by means of (Constraint) Logic Programming in mind.

2.1 Occurrence lists

On the implementation level of a solver, we can imagine quantifiers as abstract data types with components as the defining expression in the case of dependent variables, and the list of occurrences in defining expressions in the case of independent variables. These occurrence lists are optimal in the sense that they *exactly* provide the information that is needed to remove a quantifier in the back substitution phase. The maintenance of exact occurrence lists has its price, however: The occurrence list of a quantifier that is about to be back substituted is used to map over the dependent quantifiers that are affected. Afterwards we do not need the list and we can deallocate it. The main computational burden stems from the maintenance of occurrence lists of the remaining independent quantifiers. Check the following (CLP($\Re$)) example:

Before back substitution of A	occ(X)	occ(A)	occ(B)	occ(C)
$X = 6 * A - 3 * B + 10$ $A = B/2 + C + 1$	{}	{X}	{A,X}	{A}
After back substitution of A				
$X = 6 * C + 16$ $A = B/2 + C + 1$	{}	{}	{A}	{A,X}

From the example we see that occurence lists change in the course of back substitutions. One suitable data structure to cope with this situation are hash tables, yielding $O(n)$ complexity for n update operations. Existing implementations, like [Heintze et al. 91], use unordered, linked lists with $O(n^2)$ complexity.

2.2 Abstraction towards an object oriented representation of equation systems

If we consider individual quantifiers as trivial equation systems in solved form, we can abstract the process of equation solving from the object domain into a space where we speak about the unification of equation systems, where every quantifier is member of one and only one equation system. This defines an equivalence relation over quantifiers that partitions the quantifiers into equivalence classes. If one or more quantifiers are related through an equation on the object domain, we combine the corresponding equation systems into one. The maintainance of equivalence relations is of $O(n)$ complexity if we apply the well known union-find algorithm [Aho et al. 83]. Given this optimal way to maintain the equivalence relation, we can enhance the data structures that encode the equivalence relation with additional attributes that facilitate the process of equation solving on the object domain. This construction leads to an object oriented approach for the representation and manipulation of equation systems. We can, for example, chose a representation for an equation system that is particulary well suited to its size or to other properties.

Note that in order not to spoil the $O(n)$ complextity of the whole scheme, we have to ensure that combining the additional attributes of two equivalence classes is of $O(1)$. We repeat our example:

Event	Equivalence classes
	{X}, {A}, {B}
$X = 6 * A - 3 * B + 10$	
	{X,A,B}
$A = B/2 + C + 1$	
	{X,A,B,C}

The left column in the table lists a sequence of equations. The column to the right shows the equivalence classes and their members prior and after the execution of a single equation.

Assume that the list of dependent members is one of the attributes of an equivalence class. This provides the information we need for the back substitution phase. The list contains all dependent quantifiers which can possibly be affected by a back substitution. In this respect it is an approximated occurence list for *all* quantifiers. The approximation abstracts from the fact that an independent quantifier may not occur in the defining expressions of *all* dependent quantifiers. Consequently, we will end up trying to substitute quantifiers into rows where they do not occur. This sounds worse than it turns out to be in practice. In the CLP($\Re$) case we perform one back substitution step by finding the coefficient of the variable in a row pointed at by an occurence. The definition of the variable, multiplied by the coefficient, replaces the variable in the row. Given that the rows are encoded as ordered lists of variable-coefficient pairs, finding the coefficient of a variable is of linear complexity. Detecting that a variable is not member of such a list is of the same complexity. Therefore, the price we pay for the use of the approximation in the back substitution of a variable X is $O(n(m-|occ(X)|))$ where n is the dimension of the system, i.e. the number of variables, m is the number of dependent variables, i.e. the number of equations, and $|occ(X)|$ is the cardinality of the set of *actual* occurences of X. On the other hand, the approximation safes the maintainance of exact occurence lists. From empirical evidence we conclude that the savings compensate for the false drops in the back substitution phase. It is clear that we can construct examples which produce the worst case behavior for both schemes. The advantage of the use of the equivalence class abstraction is that the reference to destructive operations is reduced, which is important if we implement solvers in logical programming languages. Note that this abstraction scheme does not eliminate destructive operations required at the object level, and that the linear complexity union-find algorithm rests upon path compression — a destructive operation in lowest implementation terms. In the next section we deal with this residual destructivity in the context of a logical, unification based framework, but first let us have a look at a generalization into another direction.

2.2.1 Solved form for inequalities

The proposed equivalence class abstraction also applies to the realization of decision algorithms for systems of linear inequality relations over real valued quantifiers. This is because there is a solved form for systems of inequalities. The solved form is produced through the application of the Simplex algorithm [Dantzig 63]. The basic steps of the algorithm, the replacement of a dependent quantifier (a basic variable in linear programming terminology) by an independent quantifier (the pivot operation), is again nothing but what we called back substitution step above. The intention behind the transformation of the system into a solved form is also the same as above: Simple, operational steps allow for the decision of satisfiability and the construction of solutions. Useful attributes of an equivalence class in this context are whether there are any inequalities associated with an equation system at all, and if so, we can store redundant global information that helps the execution of the simplex algorithm.

3 Destruction is alien to Logic

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 through the convention that one particular argument of a structure representing an object and its properties is a free variable, which will eventually be bound to another structure, obeying the same convention. Therefore, sequences of modifications lead to chains of structures. The current 'value' of an object is to be found at the end of the chain. This idea carries over to *metastructures* [Neumerkel 90] and *attributed variables* [Hoitouze 90], two data types that aim at the extension of logic programming languages. Metastructures are ordinary, non-variable Prolog terms with the sole difference that they can be detected as members of this special sort. The behavior of metastructures during unifications can be specified precisely through a Prolog meta interpreter which makes unification explicit.

A comparison of the data types with regard to CLP language implementations can be found in [Holzbaur 92]. In [Holzbaur 90], the traversal of the update chains has been made transparent by moving it into the specification of metastructures and their treatment during unification. In particular, the permanent compression of the chains, as opposed to compression at garbage collection time only [Hoitouze 90], was suggested and implemented.

In an extended Prolog system following this specification, the maintainance of attributed equivalence classes is logically sound, declarative, and operationally efficient.

For the implementation of CLP languages this means that we can represent quantifiers and the equivalence classes with regard to equation systems in a logical framework: A quantifier if represented as a metastructure with some attributes. This allows for the recognition of quantifiers during unifications and for updates of the attributes.

One of the attributes of a quantifier is the equivalence class the quantifier belongs to. If two or more quantifiers are related through an equation, we simply unify the equivalence classes. As the equivalence classes are encoded via metastructures, the semantics of the unification between two objects of this sort is specified by Prolog predicates. Let us again refer to a CLP($\Re$) example:

> Assume that X and Y are the only members of their equivalence classes, prior to the submission of the equation $2 * X = 3 * Y$ to the solver. Therefore, each of the quantifiers will be represented by the following metastructures:
>
> ```
> X = t( ..., Ex),   Ex = e( ..., [X|Xt], Xt)
> Y = t( ..., Ey),   Ey = e( ..., [Y|Yt], Yt)
> ```
>
> The '...´ denote some irrelevant attributes, Ex and Ey are the metastructures representing the equivalence classes of X and Y respectively. As outlined in

an earlier section, in order to retain the linear complexity of the equivalence class maintainace algorithm, combining attributes of equivalence classes must be of $O(1)$. This is why we use difference lists [Sterling & Shapiro 86] to encode the set of dependent quantifiers in an equivalence class. After the equivalence classes are combined, and assuming that X got expressed in terms of Y, we have the following situation:

```
X = t( ..., Exy),   Exy = e( ..., [X|Xt], Xt)
Y = t( ..., Exy),
```

Now both quantifiers are members of the same equivalence class. The only dependent quantifier of this class is X. Following the specification from [Holzbaur 90], unification of equivalence classes would be implemented as:

```
meta_meta_unify( e( X, A, At, ...), e( X, B, Bt, ...)) :-
  At = B,                % append lists of dep. quants
  X = e( _, A, Bt, ...). % new equivalence class
```

4 Summary

Explicit equivalence classes of quantifiers as described above are the basis for our *Prolog* implementation of CLP($\Re$). The system solves linear equations over rational *or* real valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects implied equations, and provides for linear optimization. The mere existence of this system[3] proves that it is indeed possible to write solvers of this kind in logic programming languages. The key to such an implementation is the careful selection and introduction of suitable abstract data types.

Acknowledgements

This work was supported by the Austrian Federal Ministry of Science and Research.

References

[Aho et al. 83] Aho A.V., Hopcroft J.E., Ullman J.D.: *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[3] It is available via ftp

[Batut et al. 91] Batut C., Bernardi H., Cohen H., Olivier M.: User's Guide to PARI-GP, UFR de Mathematiques et Informatique, Universite Bordeaux, 1991.

[Boole 47] Boole G.: *The Mathematical Analysis of Logic*, Macmillan, 1947.

[Cohen 90] Cohen J.: Constraint Logic Programming Languages, Communications of the ACM, 33(7), 52-68, 1990.

[Colmerauer 90] Colmerauer A.: An Introduction to Prolog III, Communications of the ACM, 33(7), 69-90, 1990.

[Dantzig 63] Dantzig G.B.: *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[Heintze et al. 87] Heintze N., Michaylov S., Stuckey P.: CLP(R) and Some Electrical Engineering Problems, in Lassez J.L.(ed.), Logic Programming - Proceedings of the 4th International Conference - Volume 2, MIT Press, Cambridge, MA, 1987.

[Heintze et al. 91] Heintze N., Jaffar J., Michaylov S., Stuckey P., Yap R.: The CLP(R) Programmer's Manual, Version 1.1, IBM Research Division, T.J.Watson Research Center, Yorktown Heights, N.Y. 10598, 1991.

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

[Holzbaur 92] Holzbaur C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification, Oesterreichisches Forschungsinstitut fuer Artificial Intelligence, Wien, TR-92-??, 1992.

[Hoitouze 90] Huitouze S.le: A new data structure for implementing extensions to Prolog, in Deransart P. and Maluszunski J.(eds.), *Programming Language Implementation and Logic Programming*, Springer, Heidelberg, 136-150, 1990.

[Jaffar et al. 86] Jaffar J., Lassez J.L., Maher M.: A Logic Programming Language Scheme, in Groot D.de and Lindstrom G.(eds.): Logic Programming: Relations, Functions and Equations, Prentice Hall, pp.441-468, 1986.

[Jaffar et al. 91] Jaffar J., Michaylov S., Yap R.: A Methodology for Managing Hard Constraints in CLP Systems, in Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation, Toronto, Canada, 1991.

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

[Sterling & Shapiro 86] Sterling L., Shapiro E.: *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

[Tarski 48] Tarski A.: *A Decision Method for Elementary Algebra and Geometry*, University of California Press, Berkeley, CA, 1948.

[Taylor 91] Taylor A.: High Performance Prolog on a RISC, in Special Issue: Selected Papers from the Seventh International Conference on Logic Programming, New Generation Computing, 9(3,4), 1991.

A Evaluation

The performance evaluation of our proposal is presented in the appendix because we do not want to create the wrong impression that speed or speed comparison is our main concern. The primary motivation for building solvers in (C)LP languages are the same that founded and contributed to the prosperity of logic. The sole purpose of this section is therefore nothing more than to provide some evidence that our investigations are not totally detached from reality.

We report on some experiments with our implementations of CLP($\Re$) and CLP($\mathcal{Q}$) on top of SICStus Prolog version 2.1, executed on a HP/Apollo DN5500 (68040) machine. The solvers have been applied to the following examples:

1. Solve an instance of the Dirichlet problem for Laplace's equation using Liebman's five-point finite difference approximation [Heintze et al. 87]. With the data from the reference this produces a sparse equation system with 81 equations in 81 variables.

2. The matrix multiplication program from figure 2 has been applied in the 'reverse' direction to invert a Hilbert matrix of order $n = 20$, $a_{ij} = \frac{1}{i+j-1}$,

   ```
   ?- hilbert(20,H),identity(20,I),matmul(H,Inv,I).
   ```

   resulting in a sparse system of 400 equations in 400 variables. Note that inverting *large* matrices with this program is not a good idea — This is what makes a benchmark a benchmark.

3. The dense system $Ax = b$ with $a_{ij} = i^j \bmod 101$ and $b =< 1, 0 \ldots 0 >^T$ for some fixed size n of the square matrix A was solved.

Before we have a look at the actual execution times, we check in which parts of the implementation the time is spent (Table 1): The distribution clearly shows that Prolog

CLP($\Re$) Example	Low	Backsubst	ApplProg
1	76%	23%	1%
2	78%	18%	4%
3, $n = 20$	90%	9%	1%

Legend:	
Low	Low level operations like vector addition
Backsubst	Back substitutions, i.e, everything corresponding to destructive updates in other implementations
ApplProg	Execution of the pure Prolog part of the example

Table 1: Execution time distribution with CLP($\Re$)

implementations of CLP($\Re$) will benefit most from faster vector operations. If we compute with rational numbers (unlimited precision), low level numerical operations dominate 95% of the execution (Table 2): The message to Prolog implementors and vendors is clear: If Prolog should be taken serious with regard to numerical applications, we need a full[4] set of efficient numerical functions!

Table 3 relates the execution times of clpr1.1 [Heintze et al. 91] with that of our CLP($\Re$) implementation. PARI [Batut et al. 91] provides a reference point for the CLP($\mathcal{Q}$) implementation. Our programs and clpr1.1 were run on exactly the same machine (Apollo DN5500), the PARI package was timed on a SUN4/IPC, a machine approximately 10% slower than the DN5500. We assume that the reader is familiar with clpr1.1. A few words about PARI: It is a very powerful numerical and algebraic evaluator aimed at number theorists. It contains a kernel entirely written in assembly language. We used it in this comparison as the ultimate reference with regard to rational computations. The PARI code for example 3 is in figure 3.

Both CLP($\Re$) realizations produce numerically completely useless results for example 2 — Hilbert matrices are ill conditioned, and order $n = 20$ is sufficient to render double precision floating point operations meaningless. Example 3 was also not intended to be run by CLP($\Re$). We included the results for real valued computations to give an impression of the prize for ultimate precision, and to play a fair game: Experiments 1 and 2 would suggest that the Prolog version of CLP($\Re$) is slower than the clpr1.1 version by a factor

[4]Look at CommonLisp

CLP($\mathcal{Q}$) Example	Rateval	Low	Backsubst	ApplProg
2	72%	23%	4%	1%

Legend:	
Rateval	Rational scalar arithmetic
Low	Low level operations like vector addition
Backsubst	Back substitutions, i.e, everything corresponding to destructive updates in other implementations
ApplProg	Execution of the pure Prolog part of the example

Table 2: Execution time distribution with CLP($\mathcal{Q}$)

Example	clpr1.1	Prolog CLP($\Re$)	Prolog CLP($\mathcal{Q}$)	Pari1.36
1	0.067	0.6	3.9	
2	6.630	12.3	51.5	
3, $n = 20$	0.023	0.4	8.2	5.4
3, $n = 100$	1.483	39.1	9449.5	6386.2

Table 3: Various examples, execution times in seconds

```
matmul( [],    _, []).
matmul( [H|T], B, [H1|T1]) :-
  rowmul( B, H, H1),
  matmul( T, B, T1).

rowmul( [],    _,  []).              vecmul( [],      [],      S, S).
rowmul( [H|T], AV, [H1|T1]) :-       vecmul( [H1|T1], [H2|T2], In, Out) :-
  vecmul( AV, H, 0, H1),               vecmul( T1, T2, In+H1*H2, Out).
  rowmul( T, AV, T1).
```

Figure 2: Matrix multiplication in CLP($\Re$)

```
h100 = matrix(100,100,j,k,j^k%101)
v100 = vector(100,k,0)
v100[1] = 1
gauss(h100,v100)
```

Figure 3: The PARI code for example 3 with $n = 100$

between 2 and 9[5]. The factor 26 from example 3 is somewhat annoying. The problem with comparisons of this kind is that Prolog implementations depend very heavily on the current state of Prolog compiler technology — An area where advances are more likely to be expected than in 'saturated' domains like the compilation of procedural languages. [Taylor 91] reports on a comparison between his compiler and that of SICStus Prolog. The speedup on the benchmarks ranges from 12 to 42, with a mean of 24. Although the impact on our CLP instances has not been determined yet, we think that performance considerations should not dominate or suffocate the discussion of the proposed approach to CLP language implementations.

[5]This range seems to apply in the average case, however