A Real Implementation for Constructive Negation

Susana Muñoz Juan José Moreno-Navarro
`susana@fi.upm.es` `jjmoreno@fi.upm.es`

LSIIS, Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo s/n Boadilla del Monte
28660 Madrid, Spain [**]

Keywords Constructive Negation, Negation in Logic Programming, Constraint Logic Programming, Implementations of Logic Programming.

Introduction Logic Programming has been advocated as a language for system specification, especially for those involving logical behaviours, rules and knowledge. However, modelling problems involving negation, which is quite natural in many cases, is somewhat limitated if Prolog is used as the specification/implementation language. These restrictions are not related to theory viewpoint, where users can find many different models with their respective semantics; they concern practical implementation issues. The negation capabilities supported by current Prolog systems are rather limited, and there is no correct and complete implementation. Of all the proposals, constructive negation [1, 2] is probably the most promising because it has been proven to be sound and complete [4], and its semantics is fully compatible with Prolog's. Constructive negation was, in fact, announced in early versions of the Eclipse Prolog compiler, but was removed from the latest releases. The reasons seem to be related to some technical problems with the use of coroutining (risk of floundering) and the management of constrained solutions.

Our goal is to give an algorithmic description of constructive negation, i.e. explicitly stating the details needed for an implementation. We also intend to discuss the pragmatic ideas needed to provide a concrete and real implementation. Early results for a concrete implementation extending the Ciao Prolog compiler are presented. We assume some familiarity with constructive negation techniques and Chan's papers.

Constructive Negation When we tried to implement constructive negation algorithm we came across several problems, including the management of constrained answers and floundering. It is our befief that these problems cannot be easily and efficiently overcome. Therefore, we decided to design an implementation from scratch. One of our additional requirements is that we want to use a standard Prolog implementation (to be able to reuse thousands of existing Prolog lines and maintain their efficiency), so we will avoid implementation-level manipulations.

Intuitively, the constructive negation of a goal, $cneg(G)$, is the negation of the frontier $Frontier(G) \equiv C_1 \vee \ldots \vee C_N$ (formal definition in [4]) of the goal G.

[**] This work was partly supported by the Spanish MCYT project TIC2000-1632.

The solutions of $cneg(G)$ are the solutions of the combination (conjunction) of one solution of each of the negations of the N conjunctions C_i of the frontier. The negation of a single conjunction C_i is done in two phases: *Preparation* and *Negation of the formula*. We describe in detail both phases including unclear steps of the algorithm.

We provide an additional step of **simplification of the conjunction**. If one of the terms of C_i is trivially equivalent to $true$ (e.g. $X = X$, $\forall X.s(X) \neq 0$), we can eliminate this term from C_i. Symmetrically,if one of the terms is trivially $fail$ing (e.g. $X \neq X$, $\forall X.X \neq Y$, $\forall X.X \neq Y$, $\forall X.X = Y$), we can simplify $C_i \equiv fail$. The simplification phase can be carried out during the generation of frontier terms. We should take into account terms with universally quantified variables (that were not taken into account in [1, 2]) because without simplifying them it is impossible to obtain results.

We also provide a variant in the **negation of** $\overline{D}_{exp} \wedge \overline{R}_{exp}$ that is the step where the disequalities with free variables, $\overline{D}_{exp}$ and the rest of terms of C_i with free variables, $\overline{R}_{exp}$ are negated. This conjunction cannot be disclosed because of the negation of $\exists\, \overline{V}_{exp}.\, \overline{D}_{exp} \wedge \overline{R}_{exp}$, where $\overline{V}_{exp}$ gives universal quantifications: $\forall\, \overline{V}_{exp}.\; cneg(\overline{D}_{exp} \wedge \overline{R}_{exp})$. The entire constructive negation algorithm must be applied again. Variables of $\overline{V}_{exp}$ are considered as free variables. When solutions of $cneg(\overline{D}_{exp} \wedge \overline{R}_{exp})$ are obtained some can be rejected: solutions with equalities with variables in $\overline{V}_{exp}$. If there is a disequality with any of these variables, e.g. V, the variable will be universally quantified in the disequality. This is the way to negate the negation of a goal, but there is a detail that was not considered in former approaches and that is necessary to get a sound implementation: the existence of universally quantified variables in $\overline{D}_{exp} \wedge \overline{R}_{exp}$ by the iterative application of the method. So, what we are really negating is a subgoal of the form: $\exists.\, \overline{V}_{exp}\; \overline{D}_{exp} \wedge \overline{R}_{exp}$. Its negation is $\forall\, \overline{V}_{exp}.\; \neg\, (\overline{D}_{exp} \wedge \overline{R}_{exp})$.

An instrumental step for managing negation is to be able to handle disequalities between terms such as $t_1 \neq t_2$. The typical Prolog resources for handling these disequalities are limited to the built-in predicate `/== /2`, which needs both terms to be ground because it always succeeds in the presence of free variables. It is clear that a variable needs to be bound with a disequality to achieve a "constructive" behaviour. Moreover, when an equation $X = t(\overline{Y})$ is negated, the free variables in the equation must be universaly quantified, unless affected by a more external quantification, i.e. $\forall\, \overline{Y}.\; X \neq t(\overline{Y})$ is the correct negation. As we explained in [3], the inclusion of disequalities and constrained answers has a very low cost.

Optimizing the algorithm and the implementation Our constructive negation algorithm and the implementation techniques admit some additional optimizations that can improve the runtime behaviour of the system. Basically, the optimizations rely on the compact representation of information, as well as the early detection of successful or failing branches.

- **Compact information**. In our system, negative information is represented quite compactly, providing fewer solutions from the negation of $\bar{I}$. The advantage is twofold. On the one hand constraints contain more information and failing branches can be detected earlier (i.e. the search space could be smaller). On the other hand, if we ask for all solutions using backtracking, we are cutting the search tree by offering all the solutions together in a single answer.

- **Pruning subgoals**. The frontiers generation search tree can be cut with a double action over the ground subgoals: removing the subgoals whose failure we are able to detect early on, and simplifying the subgoals that can be reduced to true.
- **Constraint simplification**. During the whole process for negating a goal,the frontier variables are constrained. In cases where the constraints are satisfiable, they can be eliminated and where the constraints can be reduced to fail, the evaluation can be stopped with result *true*.

Experimental results We have firstly measured the execution times in milliseconds for the above examples when using negation as failure ($naf/1$) and constructive negation ($cneg/1$). All measurements were made using Ciao Prolog.

Using **naf** instead of **cneg** results in small ratios around 1.06 on average for ground calls with few recursive calls. So, the possible slow-down for constructive negation is not so high as we might expect for these examples. Furthermore, the results are rather similar. But the same goals with data that involve many recursive calls yield ratios near 14.69 on average w.r.t **naf**, increasing exponentially with the number of recursive calls. There are, of course, many goals that cannot be negated using the *naf* technique and that are solved using constructive negation.

Conclusion and Future Work After running some preliminary experiments with the constructive negation technique following Chan's description, we realized that the algorithm needed some additional explanations and modifications.

Having given a detailed specification of algorithm in a detailed way we proceed to provide a real, complete and consistent implementation. The result, we have reported are very encouraging, because we have proved that it is possible to extend Prolog with a constructive negation module relatively inexpensively. Nevertheless, we are working to improve the efficiency of the implementation. This include a more accurate selection of the frontier based on the demanded form. Other future work is to incorporate our algorithm at the WAM machine level.

We are testing the implementation and trying to improve the code, and our intention is to include it in the next version of Ciao Prolog [1].

References

1. D. Chan. Constructive negation based on the complete database. In *Proc. Int. Conference on LP'88*, pages 111–125. The MIT Press, 1988.
2. D. Chan. An extension of constructive negation and its application in coroutining. In *Proc. NACLP'89*, pages 477–493. The MIT Press, 1989.
3. S. Muñoz and J. J. Moreno-Navarro. How to incorporate negation in a prolog compiler. In E. Pontelli and V. Santos Costa, editors, *2nd International Workshop PADL'2000*, volume 1753 of *LNCS*, pages 124–140, Boston, MA (USA), 2000. Springer-Verlag.
4. P. Stuckey. Negation and constraint logic programming. In *Information and Computation*, volume 118(1), pages 12–33, 1995.

[1] http://www.clip.dia.fi .upm.es/Software