Constructive Negation and Constraints

Roman Barták*

Department of Theoretical Computer Science, Charles University
Malostranské nám. 2/25, Praha 1, Czech Republic

e-mail: bartak@kti.mff.cuni.cz
URL: http://kti.ms.mff.cuni.cz/~bartak/
phone: +420-2 2191 4242
fax: +420-2 2191 4323

Abstract: Inclusion of negation into logic programs is considered traditionally to be painful as the incorporation of full logic negation tends to super-exponential time complexity of the prover. Therefore the alternative approaches to negation in logic programs are studied and among them, the procedural negation as failure sounds to be the most successful and the most widely used. However, Constraint Logic Programming (CLP) is offering a different approach called constructive negation, that is becoming more popular.

In this paper we present a constructive approach to negation in logic programs. We concentrate on implementation aspects of constructive negation here, i.e., on the design of CLP(H) system, where H is the Herbrand Universe.

Keywords: constructive negation, logic programming, constraints, CLP

1 Introduction

Logic programming, i.e., programming using definite clauses, does not allow negated goals in the bodies of clauses. Also, it is known that incorporation of full logic negation tends to super-exponential complexity of the prover. However, the inclusion of some form of negation is required from the programming point of view and, thus, the alternative approaches, mostly based on Reiter's Closed World Assumption originated in databases, have been proposed.

Currently the most successful approach to negation in logic programming is procedural negation as failure which is also a part of ISO standard of Prolog. The operational behaviour of this form of negation can be easily described by the following Prolog program:

```
not P:-P,!,fail.
not P.
```

The advantages of the negation as failure, or more precisely, the negation as finite failure, make it attractive especially from the programming point of view. It uses sub-derivations to determine negative goals, thus exploiting the efficiency of the underlying logic programming system, and handling "special" features of the

* Partially supported by the Grant Agency of Czech Republic under the contract No 201/96/0197.

language, e.g., cut. However, the procedural negation as failure is known to have two important drawbacks: it can be used safely on ground subgoals, and on some particular types of non-ground goals, and it cannot generate any new bindings for query variables.

To overcome the above mentioned drawbacks of negation as failure Chan [4] introduced a new concept of constructive negation that extends the negation as failure to handle non-ground negative subgoals in a constructive manner. Its name stresses the fact that this form of negation is capable of constructing new bindings for query variables.The constructive negation scheme inherits many of the advantages of negation as failure, in particular it exploits the efficiency of the underlying logic programming system, and it handles special features of the language as well. At the same time, it removes the main drawbacks of negation as failure because constructive negation can handle non-ground negative subgoals and generates new bindings for query variables.

In [12] Stuckey proposed Constraint Logic Programming (CLP) as a much more natural framework for describing constructive negation. The CLP framework was developed in [5,7] and it is counted to be the lifesaver of logic programming for real-life applications. In CLP(A) scheme, the Herbrand Universe is displaced by a particular structure A which determines the meaning of the functions and (constraint) relation symbols of the language. The constraint viewpoint of constraint logic programming is well matched with constructive negation. Not only is constructive negation easier to understand from this point view, but it gives the clean approach to negation in constraint logic programming as well. More information on CLP can be found in [3,6,7,13].

In this paper we concentrate on the implementation aspects of constructive negation. In fact, we are interested in efficient implementation of the CLP(H) scheme where H is the Herbrand Universe with equality and disequality constraints. We design the constraint solver for solving equalities and disequalities over the Herbrand Universe and we propose a filtering system to obtain relevant solutions. The combination of the constraint solver with the filtering system enables us to implement efficiently the constructive negation. The resulting system handles negation in a more natural way which was the primary goal of this work. To justify our approach, we have implemented the ideas from this paper in two software prototypes.

The paper is organized as follows. In Section 2 we give motivation of this work. In Section 3 we discuss briefly the constraint solving over the Herbrand Universe, in particular solving equalities and disequalities and filtering the acquired solution. We devote Section 4 to the practical aspects of implementation of the constructive negation. In Section 5 we give some examples to compare the negation as failure with the concept of constructive negation. We argue for constructive negation here as it returns more natural solutions and preserves the declarative character of logic programs. In Section 7 we briefly describe two software prototypes implementing constructive negation. We conclude with some final remarks and description of future research.

2 Motivation

The procedural negation as finite failure serves very well if applied to ground goals but as soon as non-ground goals appear the results are disappointing. A rather extensive literature related to this topic documents that the drawbacks of the negation as failure tend to behaviour that corrupts the declarative character of logic programs as the following example shows.

Example:

Let *P* be the following program:

```
u(a).
v(a).
v(c).
```

Now, if we solve the goal `?-not u(X),v(X)` using the program *P* and the ordinary negation as failure, we get the answer `no`. However, if the goal `?-v(X),not u(X)` is solved, the solution is `X=c`. Note, that the only difference between above two goals is the order of atomic goals so the declarative character, and thus the solution, of the goals should be the same.

The above example shows that if the negation as failure is used with non-ground goals, it could return non-intuitive solutions (for other examples see Section 5). To avoid such non-intuitive behaviour and to keep the declarative character of logic programs we shift our attention to the constructive negation which promises to handle even the non-ground negative goals correctly. However, neither the pioneering works on constructive negation [11,12] nor the recent works on CLP [3,6] provide enough details to a successful implementation of the concept of constructive negation.

3 Constraint Solving over the Herbrand Universe

The traditional drawback of logic programms and Prolog is that they cannot handle negative information in a constructive way, i.e., the disequality $X \neq Y$ can be used as a test only. Because presence of disequalities is strongly desirable in the constructive negation approach we choose the CLP(*H*), where *H* is the Herbrand Universe with equality and disequality constraints, as a natural framework for understanding and implementing constructive negation. Solving equalities displaces naturally unification there, while disequalities can appear as a result of negating the solution of the goal. Of course, there are no difficulties to allow presence of equalities and disequalities in goals and in bodies of clauses as well.

The nature of equalities and disequalities in the Herbrand Universe enables us to implement two relatively independent components of the constraint solver, the component processing equalities and the other component processing disequalities. The cooperation between these two components is following: if the component responsible for equality solving resolves successfully the set of equalities then the result, i.e., the valuation of variables, is applied to the set of disequalities and the resulting disequalities are solved in the other component of the solver. It can be shown that if any of the components fails then the system of equalities and disequalities is inconsistent. Just note, that there is no need to iterate this process as the solution of disequalities does not further influence the solution of equalities.

The easier part of the constraint solver is processing equalities as it corresponds directly to the unification which is well understood [8]. To grasp formally the process of equality solving, in [2] we introduce a normal form for system of equalities that is a conjuction of equalities in the form x=t, where x is a variable and t is a term. Then, each system of equalities can be solved by transforming to the normal form or it is found to be inconsistent.

Similarly, in [2] we define the normal form for disequality that is $[x_i]\neq[t_i]$ where $[x_i]$ is a list of variables and $[t_i]$ is a list of terms. Note, that the normal form corresponds to the disjunction of simple disequalities $x_i\neq t_i$. Again, each system of disequalities can be solved by transforming to the normal form or it is found to be inconsistent. To simplify the system of disequalities, in [2] we define the subsumption relation between diseaqualities whose application removes some unneeded disequalities in the conjunction of disequalities, e.g., X≠a subsumes f(X,b)≠f(a,X).

Example:

initial system of disequalities: X≠a & f(X,Y,a)≠f(a,b,X) & h(Y,k(X))≠h(b,g(X))

the normal form: [X]≠[a]

(because f(X,Y,a)≠f(a,b,X) is subsumed by X≠a and h(Y,k(X))≠h(b,g(X)) is a valid disequality)

In [2], we describe the algorithms for solving equalities and disequalities by transforming them into normal form. These algorithms makes the basic constraint solver in the CLP(*H*) system.

To complete the construction of the CLP(*H*) system, there remains to answer the following question:

What should be presented to the user of the system as the result of computation?

The obvious extreme answers to the above question, i.e. nothing (yes/no answer) or everything (all equalities and disequalities), are not suitable from the point of view of constructive negation [2]. Also, the approach of most Prolog systems, which return relevant (to the goal) equalities only, is not appropriate for constructive negation because the negative information is lost (see examples in Section 5).

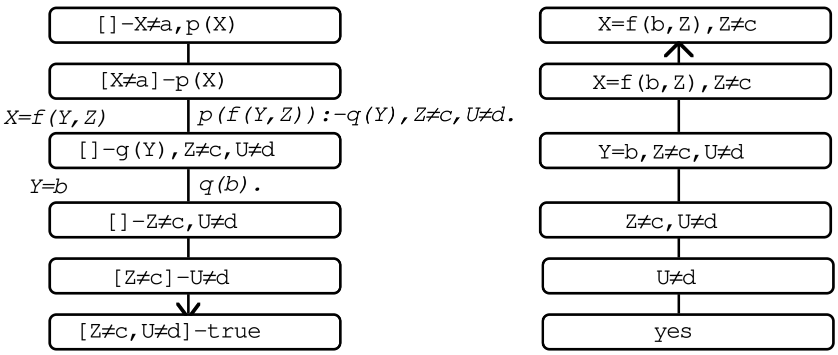
The result of above discussion is that relevant equalities and disequalities should be returned as the solution of the goal. We call the process of selecting the relevant equalities and disequalities *filtering solution*. First, the equalities relevant to the variables in the goal are selected and, then, the disequalities relevant to the variables in both the goal and the relevant equalities are selected. This is a novel approach to the definition of relevance which approved to return intutive answers as the following example shows.

Example:

Let P be the following CLP(*H*) program:

```
p(a).
p(f(Y,Z)):-q(Y),Z≠c,U≠d.
q(b).
```

The following schema captures the course of computation of the goal ?-`X≠a,p(X)` using the program P. During the computation, the satisfiable but not valid disequalities are collected to prune the further computation as you can see at the left part of the schema which shows the course of computation. We also label this part by equalities solved and by clauses used to reduce the goal. To simplify the figure, we encapsulate the standardization apart. The right part of the scheme represents the subsequent filtering of the computed solution which is displayed bellow the schema.



X=f(b,Z),Y=b,Z≠c,U≠d

4 How to Negate the Solution?

By implementing CLP(*H*), where *H* is the Herbrand Universe with equality and disequality constraints, we get the ideal framework for constructive negation. What remains is to implement the concept of constructive negation itself.

The constructive negation is based on the following procedure:

1. take a negative subgoal,
2. run the positive version of this subgoal,
3. collect all solutions of this possibly non-ground subgoal as a disjunction,
4. negate the disjunction giving a formula equivalent to the negative subgoal.

Steps 1, 2 and 3 are handled naturally by the underlying inference machine, so we have to describe only how the collected solution of the positive version of the goal is negated. Remind that the solution of the goal is a conjunction of equalities and disequalities in normal form relevant to the goal. We call such solution a *single solution*. For the constructive negation one needs to collect all single solutions of the positive version of the goal (step 3 above), so the disjunction of single solutions is constructed. We call such solution a *complete solution*. This is different from the negation as finite failure, where the existence of a single solution for the positive version of the goal implies immediately the failure of negative goal. We will discuss the efficiency of finding the complete solution later in this section.

Now, the question is how to negate the complete solution? We rely on the following formula [12] which is a property of the Herbrand Universe:

$$(\neg\exists Y,Z\ (x=t\ \&\ Q)) \Leftrightarrow (\forall Y\ (x\neq t) \vee \exists Y\ (x=t\ \&\ \neg\exists Z\ Q)) \quad \mathbf{(1)}$$

where x is a variable that does not appear neither in t nor in Q, Y is the set of variables in t (i.e., Y=vars(t)) and Z is the set of variables which appear in Q but not in t (i.e., Z=vars(Q)—Y).

The semantic meaning of the formula (1) is the following: if one uses some clause `H:-B` to solve the positive goal `?-G` and then negates the obtained solution to get a solution of the goal `?-not G` then there are two alternatives:

(i) the clause `H:-B` is prevented to be used for reduction of `G` by disabling unification of `G` and `H` (i.e., `G≠H`), or

(ii) the clause `H:-B` is used for reduction of the goal `G`, i.e., `G=H`, but the solution of `B` is negated.

These two cases correspond roughly to two elements of the disjunction in formula (1).

The following example explains the process of finding the solution of negative goal (for simplification we omit the quantifiers).

Example:

Let P be the program:

```
p(a).
p(f(Y)):-Y≠b.
```

and the goal to solve be: `?-not p(X).`

1) Run positive version of the goal: `?-p(X).`

2) Collect complete solution: $X=a \vee (X=f(Y)\ \&\ Y\neq b)$

3) Negate the complete solution: $X\neq a\ \&\ (X\neq f(Y) \vee (X=f(Y)\ \&\ Y=b))$

4) Convert to DNF and simplify: $(X\neq a\ \&\ X\neq f(Y)) \vee X=f(b)$

i.e. $\forall Y\ (X\neq a\ \&\ X\neq f(Y)) \vee \exists Y\ (X=f(b)\ \&\ Y=b)$

Note, that if we negate the acquired complete solution "mechanically", i.e., without application of the above formula (1), we get the wrong solution $(X\neq a\ \&\ X\neq f(Y)) \vee (X\neq a\ \&\ Y=b)$.

At the beginning of this section, we mentioned that finding a complete solution of the positive version of the goal can be the source of some inefficiency comparing to the negation as failure. In general, this is true but the additional information in disequalities can be exploited to prune the computational tree which subsequently speeds up the interpreter.

When a negated goal is solved, all equalities and disequalities currently collected are "frozen" and passed to the interpreter which is finding the complete solution of the positive version of the goal. This "frozen information" is used to prune the computational tree but, as the equalities and disequalities are frozen, they are not returned in the solution (and thus negated) as the following example shows.

Example:

Let procedure *p* be defined by the following two clauses:

```
p(a):-… % arbitrary body here
p(b).
```

If one solves the goal `?-X≠a,not p(X)`, the frozen disequality `X≠a` is passed to the solver when the complete solution of the goal `?-p(X)` is being computed. This prunes the computational tree, i.e., the clause `p(a):-…` is not used to reduce the goal `?-p(X)`, and the complete solution `X=b` is returned. After negation and joining with the frozen disequality `X≠a` the solution `X≠a,X≠b` of the original goal is found.

5 Examples

The original goal of this work was to implement a negation in logic programs in such a way that more intuitive solutions are produced and the declarative character of the program is preserved. The following table shows a bundle of examples and a comparison of solutions produced by the standard negation as finite failure (NF) and by our implementation of constructive negation respectively. In the "constructive negation column", the alternative solutions of the goal are depicted as individual rows.

PROGRAM	GOAL	SOLUTION	
		NF	CONSTRUCTIVE NEGATION
	`?-p(X).`	no	`X=f(Y) & Y≠a`
`p(f(Y)):-not q(Y).`	`?-not p(X).`	yes	`X≠f(Y)`
`q(a).`			`X=f(a)`
	`?-not not p(X).`	no	`X=f(Y) & Y≠a`
`s(f(Y)):-not r(Y).`	`?-s(X).`	no	no
`r(Z).`	`?-not s(X).`	yes	no
`p(a,f(Z)):-t(Z).`	`?-not p(X,Y).`	no	`X≠a & X≠f(c)`
`p(f(Z),b):-t(Z).`			`X≠f(c) & Y≠f(c)`
`t(c).`			`X≠a & Y≠b`
			`Y≠f(c) & Y≠b`
`u(a).`	`?-not(u(X),v(X)).`	no	`X≠a`
`u(b).`	`?-not u(X), not v(X).`	no	`X≠a & X≠b & X≠c`
`v(a).`	`?-not u(X), v(X).`	no	`X=c`
`v(c).`	`?-v(X), not u(X).`	`X=c`	`X=c`

Comparison - Negation as Failure vs. Constructive Negation

6 Implementation

To test ideas described in this paper we have implemented two software prototypes in Prolog based on concepts of meta-interpretation and meta-variables respectively.

Because the implementation of CLP(*H*) requires changes of the inference machine of Prolog, we use a standard technique called meta-interpretation first. We utilize the concept of extendible meta-interpreter which we proposed in our previous papers [1]. Extendible meta-interpreter is a meta-interpreter whose functionality can be extended via plug-in modules. We have implemented the constraint solver and the solution filter as such plug-in modules. The advantage of using meta-intepreters to change the standard behaviour of Prolog is that the original program and goal need not be changed. The main disadvantage of meta-intepreters is the slow down of the computation.

The second implementation utilizes the concept of meta-variables [10] and open architecture of Prolog [9]. Meta-variables are a way to extend Prolog's built-in unification by user definitions. First, we implemented a library that can be attached to arbitrary Prolog program to add functionality of meta-variables. Then, we redefined standard unification using the meta-variable concept, we implemented a disequality solver and we added a constructive negation construct `cnot`. The advantage of this approach is that the underlying Prolog interpreter is exploited as much as possible and thus the efficiency is preserved. The little drawback of this approach is that the original program and the goal have to be rewritten to use the "changed" unification and `cnot` construct.

The Prolog source code of both implementations is available on-line at URL: `http://kti.ms.mff.cuni.cz/~bartak/html/negation.html`.

7 Conclusions

In this paper we present a complete implementation of the constructive negation within the CLP(*H*) framework, where *H* is the Herbrand Universe with equality and disequality constraints. We design the constraint solver for solving equalities and disequalities over the Herbrand Universe and we also propose a filtering system to obtain relevant solutions. Finally, we exploit the foundation of CLP(*H*) to implement naturally the constructive negation. We also highlight some problems which appear during the implementation and we propose the solution of these problems.

We strongly argue for using constructive negation here as it provides more natural results than the widely used negation as failure. Also, we show that the constructive negation preserves better the declarative character of logic programs. By implementing the proposed CLP(*H*) system we prove that it is possible to incorporate constructive negation efficiently.

There is still a lot of opportunities for further research. Very interesting area is incorporation of constructive negation into CLP(*A*) over arbitrary domain *A* or into Hierarchical CLP (HCLP). Both CLP and HCLP are important from the point of view of real-life applications.

The main contribution of this work is that it shows a real implementation of constructive negation supported by the underlying theory.

Acknowledgments

I would like to thank professor Petr Štěpánek for his continuous support, useful discussions and comments on prerelease version of the paper.

References

[1] Barták, R. and Štěpánek, P., Extendible Meta-Interpreters, KYBERNETIKA, Volume 33(1997), Number 3, pp. 291-310

[2] Barták, R., Constructive Negation in CLP(H), submitted to CP'98 conference

[3] Benhamou, F. and Colmerauer, A. (eds.), *Constraint Logic Programming-Selected Research*, The MIT Press, Cambridge, Massachusetts, 1993

[4] Chan, D., Constructive Negation Based on Completed Database, in: *Proceedings of 5th International Conference on Logic Programming*, Seattle, 1988, pp. 111-125

[5] Gallaire, H., Logic programming: Further developments, in: *IEEE Symposium on Logic Programming*, pp. 88-99, IEEE, Boston, July 1985

[6] Jaffar, J., Maher, M.J., Constraint Logic Programming: A Survey, in: Journal of Logic Programming 19, pp. 503-581, 1994

[7] Jaffar, J., Lassez, J.-L., Constraint Logic Programming, in: *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111-119, Munich, Germany, January 1987

[8] Lloyd, J.W., *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984

[9] Meier, M., Schimpf, J., An Architecture for Prolog Extensions, TR ECRC-95-6, ECRC, 1995

[10] Neumerkel, U., Extensible Unification by Metastructures, in: *Proceedings of META`90*, 1990

[11] Przymusinsky, T. C., On Constructive Negation in Logic Programming, Extended Abstract, 1991

[12] Stuckey, P. J., Constructive Negation for Constraint Logic Programming, in: *Proceedings of Logic in Computer Science Conference*, 1991, pp. 328-339

[13] Van Hentenryck, P., *Constraint Satisfaction in Logic Programming*, Logic Programming Series, The MIT Press, 1989