Constructive Negation in CLP(H)

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, with the spread of Constraint Logic Programming (CLP), a different approach called constructive negation becomes more popular. The reasons for acceptance of constructive negation are the preservation of the advantages of the negation as failure, i.e., efficiency and handling special features of the language, and, at the same time, while removing the main drawbacks, i.e., handling ground negative subgoals and usage as a test only.

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. According to the CLP approach, we use equalities and disequalities to process unification and negation. We describe a constraint solver for solving equality and disequality constraints over the Herbrand Universe and we propose a unique filtering system to obtain relevant solutions. Finally, we combine the constraint solver with the filtering system to implement the constructive negation efficiently. The presented approach to constructive negation is justified by an implementation work.

Keywords: constructive negation, logic programming, Prolog, 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 a 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 language, e.g., cut. However, the procedural negation as failure is known to have two important drawbacks:

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

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 [7] introduced a new concept of constructive negation. Constructive negation 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. It is based on the following procedure:

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

This constructive negation scheme inherits many of the advantages of negation as failure. In fact, the steps 1 and 2 from the above scheme are common both to negation as failure and to constructive negation. Thus, the constructive negation 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. The steps 3 and 4 of the above procedure represent this constructive approach.

In [26] Stuckey proposed Constraint Logic Programming (CLP) as a much more natural framework for describing constructive negation. The CLP framework was developed in [12,14] 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 [6,11,13,14,27].

In this paper we concentrate on 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 unique 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. First implementation is based on the idea of extendible meta-interpreters which we proposed in our previous papers [2,3,4,5]. Second implementation utilizes the concept of meta-variables [20] which extends Prolog's built-in unification by user definitions. In the present paper, we will use notions like substitution and unification in an obvious meaning [15].

The paper is organized as follows. In Section 2 we give motivation of this work. Section 3 is dedicated to the construction of the constraint solver for solving equalities and disequalities over the Herbrand Universe. In Section 3.1 we define basic notions regarding efficient solving of equalities and disequalities and in Section 3.2 we present algorithms included in the constraint solver. In Section 4 we describe filtering system that selects solutions relevant to the original goal. We devote Section 5 to the practical aspects of implementation of the constructive negation. In Section 6 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 8 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 6). 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 [21,26] nor the recent works on CLP [6,11,13] provide enough details to a successful implementation of the concept of constructive negation.

3 Equality and disequality solver

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 CLP(*A*) system embraces primarily the constraint solver over the domain *A*. Thus, the first step in the design of the CLP(*H*) system with constructive negation is to implement equality and disequality solver over the Herbrand Universe.

3.1 Theoretical background

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 [15]. To grasp formally the process of equality solving we introduce three categories of equalities, i.e., valid, invalid and satisfiable equalities, and we define a normal form for system of equalities. Consequently we describe the process of solving systems of equalities as a transformation to the normal form (the solved system of equalities) which will be the result of equality solving.

Definition 1: (classification of equalities)

We classify equalities into following three categories (types):

a) t=u is *valid* iff $\forall\sigma$ $t\sigma\equiv u\sigma$ (i.e., $t\equiv u$)

b) t=u is *invalid* iff $\forall\sigma$ $\neg(t\sigma\equiv u\sigma)$

c) t=u is *satisfiable* iff $\exists\sigma$ $t\sigma\equiv u\sigma$ (i.e., $\neg\forall\sigma$ $\neg(t\sigma\equiv u\sigma)$)

where t, u are terms, σ is a substitution and $\equiv$ is a syntactic equality.

Example:

`f(a,Y)=f(a,Y)` is valid and satisfiable
`f(X,g(a))=f(X,b)` is invalid
`f(X,Z)=f(Y,V)` is satisfiable but not valid

According to Definition 1, each equality is either satisfiable or invalid, and some satisfiable equalities, but not all of them, are valid. Moreover, satisfiable equalities may become valid or invalid by substituting some variables while the category of both valid and invalid equalities respectively does not change by applying substitution (see above example).

The goal of equality solving is to find a (most general) substitution which, when applied to the system (conjunction) of equalities, makes the system of valid equalities. It is clear that if there is any invalid equality in the system then it is not possible to find such substitution and, thus, the system is inconsistent. Also, the inconsistency can arise from the conflict among satisfiable but not valid equalities, e.g., the system `X=a & X=b` is inconsistent.

The process of finding above described substitution is well known under the name unification and the substitution found is called (most general) unifier [15]. In the following paragraphs we remind/redefine some notions to be consistent with constraint approach where solving equalities corresponds to finding the most general unifier.

Definition 2: (unifier, inconsistency, equivalence, normal form)

1) We call the substitution σ a *unifier* for the system (conjunction) *E* of equalities if $E\sigma$ is a system of valid equalities.
2) We call the system *E* of equalities *inconsistent* if there does not exist unifier for *E*.
3) Two systems of equalities are *equivalent* if they have the same unifiers.
4) The system (conjunction) of equalities is in *normal form* if it consists of equalities $x_i=t_i$, where x_i is a variable, t_i is a term and

$\forall i,j$ $x_i \notin vars(t_j)$ & $(i\neq j \Rightarrow x_i\neq x_j)$.

The notion of unifier is well known, we only customize it to fit in the concept of solving equalities. Similarly, the notions of inconsistent and equivalent systems of equalities are obvious. Note also, that the normal form for system of equalities corresponds to the notion of substitution.

The process of solving system of equalities consists of transformation of the given system of equalities into equivalent system of equalities in normal form. We use the normal form to represent the solution of the system of equalities. The following theorem justifies the process of solving system of equalities.

Theorem 1: (handling equalities)

1) The type of equality, i.e., valid, invalid or satisfiable equality respectively, is decidable algorithmically.
2) Every satisfiable equality is transformable algorithmically to the equivalent normal form.
3) Every system E of equalities is either transformable algorithmically to the equivalent normal form or it is decidable that E is inconsistent.

Due to the space restrictions, we do not prove Theorem 1 formally here, however, we describe the algorithms mentioned in the theorem in the next section which can be seen as the proof of Theorem 1. Note also, that Theorem 1 is a consequence of the well known Unification Theorem [15].

While processing equalities is well known in logic programming, handling disequalities in a constructive manner is relatively new to this area. The disequalities which appear in various Prolog systems are processed in a similar way as the negation as failure, i.e., it can only handle ground terms and it can only be used as a test (it cannot generate any new bindings for query variables). Thus, it is possible to collect disequalities only and use them as a test as soon as the disequality becomes ground. However, in the constructive negation we prefer the disequalities to behave constructively, i.e., they reduce domains of variables. So, we propose a solver that simplifies disequalities and detects the validity of the disequality as soon as possible.

Similarly to equalities, we classify the disequalities into three categories.

Definition 3: (classification of disequalities)

We classify disequalities into following three categories (types):

a) $t \neq u$ is *valid* iff $\forall\sigma\ \neg(t\sigma \equiv u\sigma)$

b) $t \neq u$ is *invalid* iff $\forall\sigma\ (t\sigma \equiv u\sigma)$ (i.e., $t \equiv u$)

c) $t \neq u$ is *satisfiable* iff $\exists\sigma\ \neg(t\sigma \equiv u\sigma)$ (i.e., $\neg\forall\sigma\ t\sigma \equiv u\sigma$)

where t, u are terms, σ is a substitution and $\equiv$ is a syntactic equality.

Example:

`f(X,g(a))≠f(X,b)` is valid
`f(a,Y)≠f(a,Y)` is invalid
`f(X,Z)≠f(Y,V)` is satisfiable but not valid

Again, according to Definition 3, each disequality is either satisfiable or invalid, and some satisfiable disequalities, but not all of them, are valid. Moreover, satisfiable disequalities may become valid or invalid by substituting some variables while the category of both valid and invalid disequalities respectively does not change by applying substitution (see above example).

Similarly to the case of equalities, the process of solving system of disequalities consists of transformation of the given system of disequalities into equivalent system of disequalities in normal form. The definition of the normal form is a bit complicated for disequalities. Nevertheless, the notions of unifier, inconsistency and equivalence can be naturally extended to disequalities. We can also define the relation of subsumption that helps one to simplify the system of disequalities.

Definition 4: (unifier, inconsistency, equivalence, subsumption of disequalities)

1) We call the substitution σ a *unifier* for the system (conjunction) *DE* of disequalities if $DE\sigma$ is a system of valid disequalities.
2) We call the system *DE* of disequalities *inconsistent* if there does not exist unifier for *DE*.
3) Two systems of disequalities are *equivalent* if they have the same unifiers.
4) We say that the system of disequalities DE1 *subsumes* the system of disequalities DE2 iff every unifier of DE1 is also a unifier of DE2.

Example:

1) `{X/a,Y/b}` is one of unifiers for disequality `f(X,Z)≠f(Y,V)`
2) the system `f(a,Y)≠f(a,Y) & Y≠b` is inconsistent
3) `f(X)≠f(a)` is equivalent to `X≠a`
4a) `X≠f(Y)` subsumes both `g(X,b)≠g(f(Y),Z)` and `f(Y)≠X`
4b) be careful, `X≠f(Y)` does not subsume `X≠f(a)` (unifier `{X/f(a),Y/b}`) and `X≠f(a)` does not subsume `X≠f(Y)` (unifier `{X/f(b),Y/b}`)

Note, that even the definitions of unifier and inconsistency respectively looks similar for equalities and disequalities, there are some differences between them. While the unifier for equalities can be constructed using the structures from the original equalities, the unifier for disequalities requires some additional term structures/constants (see above example). Nevertheless, this is not a problem as we really do not need to construct the unifier for disequalities (opposite to equalities, where the most general unifier is constructed). We use the notion of unifier to define the equivalence among disequalities and the subsumption relation only.

Also, the notion of inconsistent system (conjunction) of disequalities is a bit simpler as there are no cross-links among disequalities in the system of disequalities. Clearly, the system (conjunction) of disequalities is inconsistent if and only if this system contains at least one invalid disequality. This is a consequence of the above observation on construction of a unifier for disequalities.

The notions of equivalence and subsumption are important for solving the system of disequalities. Especially the subsumption can simplify markedly the system of disequalities as the following theorem shows.

Theorem 2: (subsumption and equivalence)

1) Two systems of disequalities DE1 and DE2 are equivalent if and only if DE1 subsumes DE2 and DE2 subsumes DE1.
2) Let the system of disequalities DE1 subsume the other system of disequalities DE2 and DE be the system of disequalities consisting of disequalities from DE1 and DE2 only, then DE is equivalent to DE1.

The proof of Theorem 2 is a direct consequence of Definition 4 and the observation that the set of unifiers for system (conjunction) of disequalities is equal to the intersection of sets of unifiers for individual disequalities from the system.

Example:

`X≠a & f(X,g(Y))≠f(a,V))` is equivalent to `X≠a`

According to Theorem 2, the subsumption relation enables simplification of the system of disequalities. However, we also require further simplification of individual disequalities, e.g., `h(X)≠h(g(Y))` can be simplified into equivalent form `X≠g(Y)` in the Herbrand Universe. Therefore, we introduce a normal form of disequality.

Definition 5: (normal form of disequality)

1) We say that the disequality A≠B is in *normal form* if A is a list $[x_i]$ of n variables ($n \geq 1$), B is a list $[t_i]$ of n terms, $\forall i\ x_i \notin vars(t_i)$ and $x_i \neq t_i$ does not subsume $x_j \neq t_j$ ($i \neq j$). If $n=1$, then we speak about simple normal form and we simply write $x_1 \neq t_1$ instead of $[x_1] \neq [t_1]$

2) Similarly, we say that the system of disequalities is in *normal form* if each disequality in the system is in normal form and none disequality from the system subsumes other disequality from the system.

Example:

1) `h(Y,k(g(X),Y,X))≠h(X,k(g(f(a)),g(Z),Y))` has normal form
`[Y,X,Y]≠[X,f(a),g(Z)]`
2) `X≠a & f(a)≠f(X) & f(Z)≠f(g(X))` has normal form `X≠a & Z≠g(X)`

Similarly to equalities, we can transform algorithmically the system of disequalities into equivalent normal formal. This transformation makes the kernel of the disequality solver and, again, we use the normal form to represent the solution of the system of disequalities. The following theorem justifies the process of solving the system of disequalities.

Theorem 3: (handling disequalities)

1) The type of disequality, i.e., valid, invalid or satisfiable disequality respectively, is decidable algorithmically.
2) Every satisfiable disequality is transformable algorithmically to the equivalent normal form.
3) Every system DE of disequalities is either transformable algorithmically to the equivalent normal form or it is decidable that DE is inconsistent.

Due to the space restrictions, we do not prove Theorem 3 formally here, however, we present the algorithms mentioned in the theorem in the next section which can be seen as the proof of Theorem 3.

3.2 The algorithms

In this section we present skeletons of algorithms for solving equalities and disequalities over the Herbrand Universe. We use a pseudo-language which inherits features of procedural languages, e.g., `case` and `while` constructs, as well as unification and the `fail` construct inherited from Prolog. The occurrence of `fail` during the computation means that the computation of the solver fails, i.e., the system of equalities and disequalities is not consistent. However, there is no backtracking when failure occurs.

First, we present the equality solver which consist of procedures `solve_eq` and `solve_eq_list`. These procedures closely cooperate to solve given system of equalities. As the code is self-explanatory, we do not attach further comments. Also, we do not include code of all sub-routines like `append` or `apply_substitutions`. The procedure `make_eqs` used within the procedure `solve_eq` transfers two list of terms into a list of equalities, i.e., $[a_1,\ldots,a_n]$ and $[b_1,\ldots,b_n]$ are transferred to $[a_1=b_1,\ldots,a_n=b_n]$.

```
solve_eq(A=B,Solution)
      case of
        :var(A) & var(B)
              if A=B then
                    Solution:=[]
              else
                    Solution:=[A=B]
              end if
        :var(A) & nonvar(B)
              if not member(A,vars(B)) then
                    Solution:=[A=B]
              else
                    fail
              end if
        :nonvar(A) & var(B)
              if not member(B,vars(A)) then
                    Solution:=[B=A]
              else
                    fail
              end if
        :nonvar(A) & nonvar(B)
              if functor(A)=functor(B) then
                    make_eqs(args(A),args(B),EqsOfArgs)
                    solve_eq_list(EqsOfArgs,Solution)
              else
                    fail
              end if
      end case
end solve_eq

solve_eq_list(EqList,Solution)
      Solution:=[]
      while not empty EqList
        Eq:=delete_first(EqList)
        solve_eq(Eq,Sol1)
        EqList:=apply_substitution(Sol1,EqList)
        Solution:=apply_substitution(Sol1,Solution)
        Solution:=append(Solution,Sol1)
      end while
end solve_eq_list
```

The equality solver returns three types of solutions:

- if the system of equalities is inconsistent, then the solver fails, i.e., stops by executing the fail command,
- if all equalities are valid, then the empty list is returned,
- otherwise, the normal form of the system of equalities, which represents the most general unifier, is returned.

The following three procedures `solve_deq`, `solve_disj_deq` and `solve_deq_list` make the disequality solver. The structure of disequality solver is similar to the structure of equality solver, however, there are some differences reflecting the nature of disequalities, e.g., the special procedure `solve_disj_deq` is required. Instead of explaining these differences here, we enclose the trace of the example computation below the description of the algorithms.

Because of lack of space, we do not include code of functions `join_with_subsumption` and `add_with_subsumption`. Nevertheless, the definition of these functions can be found in the implementation. The function `join_with_subsumption` joins two disequalities in the normal form into one disequality in the normal form, e.g., X≠a, Y≠b -> [X,Y]≠[a,b]. The function `add_with_subsumption` appends the disequality in the normal form to the list (conjunction) of disequalities in the normal form, e.g., X≠a, {[X,Y]≠[a,b],Y≠c} -> {X≠a, Y≠c}. Both functions include the subsumption test that removes the subsumed (i.e., unneeded) disequalities.

```
solve_deq(A≠B,Solution)
      case of
        :var(A) & var(B)
              if A=B then
                    fail
              else
                    Solution:=(A≠B)
              end if
        :var(A) & nonvar(B)
              if member(A,vars(B)) then
                    Solution:=true
              else
                    Solution:=(A≠B)
              end if
        :nonvar(A) & var(B)
              if member(B,vars(A)) then
                    Solution:=true
              else
                    Solution:=(B≠A)
              end if
        :nonvar(A) & nonvar(B)
              if functor(A)=functor(B) then
                    solve_disj_deq(args(A),args(B),Solution)
              else
                    Solution=true
              end if
      end case
end solve_deq
```

```
solve_disj_deq(A,B,Solution)
      case of
         :empty(A) & empty(B)
               fail
         :length(A)≠length(B) then
               Solution:=true
         :otherwise
               Solution:=nothing
               while not empty(A) and Solution≠true do
                     AH:=delete_first(A)
                     BH:=delete_first(B)
                     solve_deq(AH≠BH,SolH)
                     Solution:=join_with_subsumption(Solution,SolH)
               end while
      end case
end solve_disj_deq

solve_deq_list(DeqList,Solution)
      Solution:=[]
      while not empty DeqList
         Deq:=delete_first(DeqList)
         solve_deq(Deq,Sol1)
         Solution:=add_with_subsumption(Sol1,Solution)
      end while
end solve_deq_list
```

Here is a sample trace of the disequality solver applied to the system of disequalities:

`X≠a, f(X,Y,a)≠f(a,b,X), h(Y,k(X))≠h(b,g(X)).`

We include the names of the main procedures, their input arguments and also the output. To simplify the notation, we collapse the calls to `join_with_subsumption` and `add_with_subsumption` procedures into one call at the end of each respective procedure.

```
solve_deq_list([X≠a,f(X,Y,a)≠f(a,b,X),h(Y,k(X))≠h(b,g(X))])
      solve_deq(X≠a) -> X≠a
      solve_deq(f(X,Y,a)≠f(a,b,X))
         solve_disj_deq([X,Y,a],[a,b,X])
               solve_deq(X≠a) -> X≠a
               solve_deq(Y≠b) -> Y≠b
               solve_deq(a≠X) -> X≠a
               join_with_subsumption -> [X,Y]≠[a,b]
      solve_deq(h(Y,k(X))≠h(b,g(X)))
         solve_disj_deq([Y,k(X)],[b,g(X)])
               solve_deq(Y≠b) -> Y≠b
               solve_deq(k(X)≠g(X)) -> true
               join_with_subsumption -> true
      add_with_subsumption ->[X≠a]
```

Finally, we present the algorithm of constraint solver which solves system of equalities and disequalities. The algorithm follows the idea, that the equalities are solved first, then the solution, i.e., the valuation of variables, is applied to disequalities and the resulting system of disequalities is solved. Clearly the solution of disequalities does not further influence the solution of equalities.

```
solver(System,Solution)
      distribute(System,Equalities,Disequalities)
      solve_eq_list(Equalities,EqSolution)
      DE:=apply_substitution(EqSolution,Disequalities)
      solve_deq_list(DE,DeqSolution)
      Solution:=combine(EqSolution,DeqSolution)
end solver
```

4 Filtering solutions

In the previous sections we present a constraint solver that processes equalities and disequalities over the Herbrand Universe. 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?

There are two obvious extreme answers to the above question:

1) nothing, which corresponds to the yes/no answer, or
2) everything, i.e., the solution of all equalities and disequalities which appear during the computation.

Neither the first nor the second approach is suitable from the point of view of constructive negation. If yes/no answers are presented then we get the old fashioned negation as failure. If everything is dumped to the user then he or she becomes overwhelmed by information even if a trivial goal is solved. Moreover, if such solution is further negated, some artifacts may appear as the following example shows.

Example:

Let `p(X):-Y≠2,X=3` be the only clause for `p`. Now, if one solves the goal `not p(X)`, the complete solution `X=3 & Y≠2` of the positive goal `p(X)` is negated and the disjunction `X≠3 ∨ Y=2` is acquired that corresponds to two answers: `X≠3` and `Y=2`. Clearly, the answer `Y=2` has nothing in common with the original goal `not p(X)` because the variable `Y` is neither present in the goal nor is bound with the variable `X`.

The above discussion shows that only some relevant information should be presented to the user. However, the approach of most Prolog systems, which return relevant equalities only, i.e., the valuation of variables from the original goal, is not appropriate for constructive negation because the negative information is lost. It could result in missing some solutions for negated goals as the following example shows.

Example:

Let `q(X):-X=f(Y),Y≠a` be the only clause for `q`. If a negative goal `not q(X)` is solved and only the equality `X=f(Y)` is returned as the solution of the positive version `q(X)`, then the negation of this solution is `X≠f(Y)`. Clearly, the equality `X=f(a)` is also a solution of the goal `not q(X)` and thus the solution `X≠f(Y)` is not complete.

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*.

Now, we define the notion of relevance to the set of variables for normal form of equalities and disequalities which constitute the solution. Then we show, how this definition is applied in practice.

Definition 6: (relevant equality and disequality)

1) We say that the equality x=t in normal form is *relevant* to the set of variables V if $(x\in V \vee t\in V)$.
2) We say that the disequality $[x_i]\neq[t_i]$ in normal form is *relevant* to the set of variables V if $(\forall i\ (x_i\in V \vee t_i\in V))$.

Because of different nature of equalities and disequalities, we have to define solution relevant to the goal more carefully.

Definition 7: (relevant solution)

Let V be the set of variables from the goal G, E be the set of equalities and DE be the set of disequalities from the solution of the goal G. Let ES be the set of equalities from E relevant to V, V1 be a union $V\cup vars(ES)$ and DES be the set of disequalities from DE relevant to V1. Then ES∪DES is the *solution relevant* to the goal G.

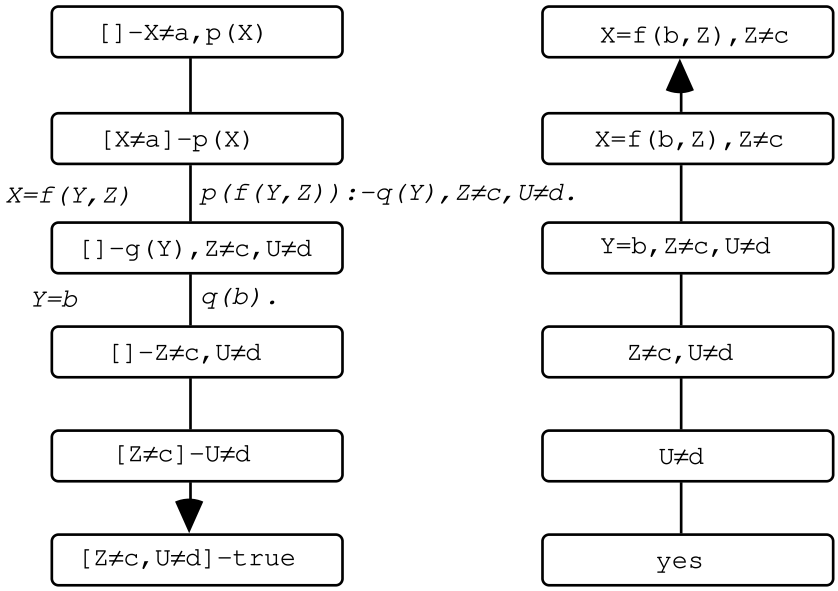
The following example shows how Definition 7 of relevant solution is applied to filter solutions in practice.

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`

5 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 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 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 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 [26] which is a property of the Herbrand Universe:

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

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 above formula is used to negate a single solution. Then, the negation of the complete solution corresponds to the conjunction of negated single solutions. Subsequently, this conjunction is converted to disjunctive normal form (DNF) in order to get the complete solution of the negative goal in appropriate form (remind, the complete solution is a disjunction of single solutions). Finally, the single solutions from this complete solution can be returned via backtracking. The following example explains the above 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 in comparison with 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 for reduction of `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.

6 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 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	
		NEGATION AS FAILURE	**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`	`yes`
`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

7 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 [1,22,23,24] first. We utilize the concept of extendible meta-interpreter which we proposed in our previous papers [2,3,4,5]. Extendible meta-interpreter is a meta-interpreter whose functionality can be extended via plug-in modules. The skeleton of such extendible meta-interpreter, we call it kernel, is as follows:

```
solve(Task,Result):-
      empty_goal(Task,Result).
solve(Task,Result):-
      select_subgoal(Task,Goal,Frontier),
      expand_goal(Goal,ExpandedGoal,Rule),
      make_task(Frontier,ExpandedGoal,NewTask),
      solve(NewTask,SubResult),
      customize_solution(Frontier,Rule,SubResult,Result).
solve(Task,Result):-
      rest_solution(Task,Result).
```

The constraint solving, i.e., the equality and disequality solver is hidden in the procedure `expand_goal`, while the solution filter is naturally represented by the procedure `customize_solution`. The implementation of these procedures is called a plug-in module or extension of the meta-interpreter. 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 [20] and open architecture of Prolog [17,18,19]. 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. However, the drawback is that the original program and 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`.

8 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 by transforming into normal form which we define here. 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 which is also presented here.

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] Abramson, H. and Rogers, M.H. (eds.), *Meta-Programming in Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1989

[2] Barták, R., Meta-interpretation of logic programs (in Czech), Diploma Thesis, Department of Theoretical Computer Science, Charles University, Prague, 1993

[3] Barták, R., A Plug-In Architecture of Constraint Hierarchy Solvers, in: *Proceedings of PACT'97*, pp. 359-371, London, 1997

[4] Barták, R. and Štěpánek, P., Meta-Interpreters and Expert Systems, Tech. Report No 115, Department of Theoretical Computer Science, Charles University, October 1995

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

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

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

[8] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, Berlin, 1981

[9] Covington, M.A., Efficient Prolog: A Practical Guide, Tech. Report AI-1989-08, The University of Georgia, August 1989

[10] Dix, J., Pereira, L.M., Przymusinsky, T.C. (eds.), Non-Monotonic Extensions of Logic Programming, Lecture Notes in Artificial Intelligence 927, Springer Verlag, Berlin, 1995

[11] Frühwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E., Wallace, M., Constraint Logic Programming - An Informal Introduction, Tech. Report ECRC-93-5, ECRC, February 1993

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

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

[14] 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

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

[16] Maher, M.J., Stuckey, P.J., Expanding Query Power in Constraint Logic Programming, in: *Proceedings of the North Americal Conference on Logic Programming*, Cleveland, October 1989

[17] Meier, M., Event Handling in Prolog, Tech. Report ECRC-95-09, ECRC, 1995

[18] Meier, M., Brisset, P., Open Architecture for CLP, TR ECRC-95-10, ECRC, 1995

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

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

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

[22] Sterling, L., Meta-Interpreters: The Flavors of Logic Programming?, in: *Proceedings of Workshop on foundation of Logic Programming and Deductive Databases*, Washington, 1986

[23] Sterling, L., Constructing Meta-Interpreters for Logic Programs, in: *Advanced School on Foundations of Logic Programming*, Alghero, Sardinia, Italy, September 1988

[24] Sterling, L. and Lakhotia, A., Composing Prolog Meta-Interpreters, in: *Proceedings of 5th International Logic Programming Conference*, Seattle, 1988

[25] Sterling, L. and Shapiro, E., *The Art of Prolog*, The MIT Press, Cambridge, Massachusetts, 1986

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

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