Efficient Implementation of General Negation Using Abstract Interpretation

Susana Muñoz Juan José Moreno Manuel Hermenegildo

Abstract

While negation has been a very active area of research in logic programming, comparatively few papers have been devoted to implementation issues. Furthermore, the negation-related capabilities of current Prolog systems are limited. We recently presented a novel method for incorporating negation in a Prolog compiler which takes a number of existing methods (some modified and improved) and uses them in a combined fashion. The method makes use of information provided by a global analysis of the source code. Our previous work focused on the systematic description of the techniques and the reasoning about correctness and completeness of the method, but provided no experimental evidence to evaluate the proposal. In this paper, after proposing some extensions to the method, we provide experimental data which indicates that the method is not only feasible but also quite promising from the efficiency point of view. In addition, the tests have provided new insight as to how to improve the proposal further. Abstract interpretation techniques (in particular those included in the Ciao Prolog system preprocessor) have had a significant role in the success of the technique.

Keywords: Negation in Logic Programming, Constraint Logic Programming, Program Analysis, Implementations of Logic Programming, Abstract Interpretation.

1 Introduction

The fundamental idea behind Logic Programming (LP) is to use a computable subset of logic as a programming language. Probably, negation is the most significant aspect of logic that was not included from the start. This is due to the fact that dealing with negation involves significant additional complexity. However, negation has an important role for example in knowledge representation, where many of its uses cannot be simulated by positive programs. Declarative modeling of problem specifications typically also include negative as well as positive characteristics of the domain of the problem. Negation is also useful in the management of databases, program composition, manipulation and transformation, and default reasoning, etc.

Universidad Politécnica de Madrid, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain, E-mail: `susana|jjmoreno|herme@fi.upm.es`.

The perceived importance of negation has resulted in significant research and the proposal of many alternative ways to understand and incorporate negation into LP. The problems involved start already at the semantic level and the different proposals (negation as failure, stable models, well founded semantics, explicit negation, etc.) differ not only in expressivity but also in semantics. Presumably as a result of this, implementation aspects have received comparatively little attention. A search on the The Collection of Computer Science Bibliographies [14] with the keyword "negation" yields nearly 60 papers, but only 2 include implementation in the keywords, and fewer than 10 treat implementation issues at all.

Perhaps because of this, the negation techniques supported by current Prolog compilers are rather limited:

- Negation as failure (sound only under some circumstances) is a built-in or library in most Prolog compilers (Quintus, SICStus, Ciao, BinProlog, etc.).
- The "delay technique" (applying negation as failure only *when* the variables of the negated goal become ground, which is sound but incomplete due to the possibility of floundering) is present in Nu-Prolog, Gödel, and Prolog systems which implement delays (most of those above).
- Constructive negation was announced in early versions Eclipse, but appears to have been removed from more recent releases.

Our objective is to design and implement a practical form of negation and incorporate it into a Prolog compiler. In [21] we studied systematically what we understood to be the most interesting existing proposals: negation as failure (*naf*) [8], use of delays to apply *naf* in a secure way [18], intensional negation [1, 2], and constructive negation [6, 7]. We could not find a single technique that offered both completeness and an efficient implementation. However, we proposed to use a combination of these techniques and that information from a static analysis of the program could be used to reduce the cost of selecting among techniques. We provided a coherent presentation of the techniques, implementation solutions, and a proof of correctness for the method, but we did not provide any experimental evidence to support the proposal. This is the purpose of this paper. We also sketch an implementation for constructive negation, which was missing from [21].

One problem that we face is the lack of a good collection of benchmarks using negation to be used in the tests. One of the reasons has been discussed before: there are few papers about implementation of negation. Another fact is that negation is typically used in small parts of programs and is not one of their main components. We have however collected a number of examples using negation from logic programming textbooks, research papers, and our own experience teaching Prolog.

We have tested these examples with all of our techniques in order to establish their efficiency. We have also measured the improvement of efficiency thanks to the use of the static analyzers. We have used the Ciao system [4] that is an efficient Prolog implementation and incorporates all the needed static analyses. However, it is important to point out that the techniques used are fairly standard, so they can be incorporated into almost any Prolog compiler.

In both cases the results have been very interesting. The comparison of the techniques has allowed us to improve the right order in which to apply them. Furthermore, we have learned that the impact of the use of the information from the analyzers is quite significant.

The rest of the paper is organized as follows. Section 2 presents more details on our method to handle negation and how it has been included in the Ciao system. Section 3 presents the evaluation of the techniques and how the results have helped us reformulate our strategy. The impact of the use of abstract interpretation is studied in 3.3.

2 Implementation of a Negation System

In this section we present the techniques from the literature which we have integrated in a uniform framework. The techniques and the proposed combination share the following characteristics:

- We are interested in techniques with a single and simple semantics. The simplest alternative is to use the Closed Word Assumption (CWA) [8] by program completion and Kunen's 3-valued semantics [12]. These semantics will be the basis for soundness results.
- Another important issue is that they must be "constructive", i.e., program execution should produce adequate goal variable values for making a negated goal false. Chan's constructive negation [6, 7] fulfills both objectives. However, it is difficult to implement and expensive in terms of execution resources. Our idea is to use the simplest technique for each particular case.
- The formulations need to be uniform in order to allow the mixture of techniques. We also need to establish sufficient correctness conditions to use them.
- We also provide a Prolog implementation of each of the techniques. This allows combining the implementations as also obtaining a portable implementation of negation.

2.1 Negation as failure and delays

Clark's negation as finite failure rule [8] states that $\neg Q$ is a consequence of a program P if there exists a finitely failed SLD tree for the query Q with respect to P (in short, if Q finitely fails). Prolog systems typically include the following implementation:

```
naf(Q) :- Q, !, fail.
naf(Q).
```

which is unsound unless the free variables of Q are constrained. A correct simplification is to apply the technique when Q has no free variables. This technique is adapted to a sound version usually by using delay directives (e.g., `when`) to ensure that the call to negation as failure is made only when the variables of the negated goal are ground. A call to $\neg p(\overline{X})$ is replaced by:

```
..., (delay (X̄), naf(p(X̄))), ...
```

2.2 Disequality constraints

An instrumental step in order to manage negation in a more advanced way is to be able to handle disequalities between terms such as $t_1 \neq t_2$. Prolog implementations typically include only the built-in predicate `≡` which can only work with disequalities if both terms are ground and simply succeedss in the presence of free variables. A "constructive" behaviour must allow the "binding" of a variable with a disequality, i.e., the solution to the goal `X ≡ t` would be the constraint $X \neq t$. This is exactly an implementation of CLP($\mathcal{H}$) (constraints over the Herbrand Universe with equality and disequality). Several CLP extensions of Prolog (Prolog III for instance) include this feature but it is not usually available in Prolog compilers. However, it can be included at a relatively low cost [21].

The starting point is an adequate representation of constraint answers. A disequation $c(X, a) \neq c(b, Y)$ produces a disjunction $X \neq b \vee Y \neq a$. So, conjunctions of disjunctions of disequations are used as normal forms. On the other hand, the negation of an equation $X = t(\overline{Y})$ produces the universal quantification of the free variables in the equation, unless a more external quantification affects them. The negation of such an equation is $\forall\, \overline{Y}\; X \neq t(\overline{Y})$. Universally quantified disequations are allowed in the constraints too. Therefore, the normal form of constraints is:

$$\underbrace{\bigwedge_i (X_i = t_i)}_{\text{positive information}} \qquad \underbrace{\wedge \bigvee_j \forall\, \overline{Z}_j^1\; (Y_j^1 \neq s_j^1) \wedge \ldots \wedge \bigvee_l \forall\, \overline{Z}_l^n\; (Y_l^n \neq s_l^n)}_{\text{negative information}}$$

where each X_i appears only in $X_i = t_i$, none s_k^r is Y_k^r and the universal quantification could be empty (leaving a simple disequality). In [21] details can be found on how this normal form is computed and preserved by unification, equalities, and disequalities. For the Prolog implementation we use attributed variables [11] (included, for instance, in SICStus Prolog or Ciao Prolog, and also in Eclipse in the form of "meta-structures"). Attributed variables are variables with an associated attribute, which is a term. They behave like ordinary variables, except that the programmer can supply code for unification, printing facilities and memory management which is called when such variables are involved. In our case, we will associate to each variable a data structure containing a normal form constraint. The main task is to provide new unification code. To this end, once the unification of a variable X with a term t is triggered, there are three possible cases (up to commutativity):

1. if X is a free variable and t is not a variable with a negative constraint, X is just bound to t;
2. if X is a free variable or bound to a term t' and t is a variable Y with a negative constraint, we need to check if X (or, equivalently, t') satisfies the constraint associated with Y;
3. if X is bound to a term t' and t is a term (or a variable bound to a term), the classical unification algorithm can be used.

A predicate `=/=`, used to check disequalities, is defined in a similar way to explicit unification (`=`). The main difference is that it incorporates negative constraints instead of bindings and the decomposition step can produce disjunctions.

The attribute/constraint of a variable is represented as a list of lists of pairs (variable, term) using a constructor `/`, i.e., the disequality $X \neq 1$ is represented as `X/1`. The outer list is used to represent a disjunction while the inner lists represent conjunctions of disequalities. When a universal quantification is used in a disequality (e.g., $\forall Y \; X \neq c(Y)$) the new constructor `fA`/2 is used (the previous constraint is represented as `fA(Y, X / c(Y))`.

2.3 Constructive negation for finite solutions

Constructive negation is generally accepted as the "right" method to handle negation (when Kunen's 3-valued semantics are used). It was proposed by Chan in two steps [6, 7], and later formalized by Stuckey [22] in the context of CLP. The first of Chan's papers presented the main idea –in order to obtain the solutions for $\neg Q$ we proceed as follows:

1. Firstly, the solutions for Q are obtained getting a disjunction: $Q \equiv S_1 \vee S_2 \vee \ldots \vee S_n$. Each component S_i can be understood as a conjunction of equalities: $S_i \equiv S_i^1 \wedge S_i^2 \wedge \ldots \wedge S_i^{m_i}$
2. Then the formula is negated and a normal form constraint is obtained:

$$\begin{array}{rll} \neg Q \equiv & \neg(S_1 \vee S_2 \vee \ldots \vee S_n) & \equiv \\ & \neg S_1 \wedge \neg S_2 \wedge \ldots \wedge \neg S_n & \equiv \\ & \neg(S_1^1 \wedge \ldots \wedge S_1^{m_1}) \wedge \ldots \wedge \neg(S_n^1 \wedge \ldots \wedge S_n^{m_n}) & \equiv \\ & (\neg S_1^1 \vee \ldots \vee \neg S_1^{m_1}) \wedge \ldots \wedge (\neg S_n^1 \vee \ldots \vee \neg S_n^{m_n}) & \end{array}$$

   The formula can be obtained in different ways depending on how we negate a solution. It can also be arranged into a disjunction of conjunctions according to the variables in each S_i^j.

However, this method is not applicable when the negated goal Q has infinitely many answers. For this reason, this paper was considered "in error" and the second one, which solves the problem, is typically used as the correct reference. However, we found this simpler finite version worth exploring because its implementation is easier than full constructive negation, and it can be used if the number of solutions can be determined to be finite. To this end, we have implemented a Prolog predicate `cnegf(Q)` to implement finite constructive negation, which works as follows:

1. First of all, all variables V of the goal Q are obtained.
2. Then, all Q's solutions for variables in V are computed using `setof`/3. Each solution is a constraint in normal form.
3. Finally, the negation of each solution is computed and combined to obtain the answers of $\neg Q$ one by one.

The last point is the most important one and several alternatives are possible. Thanks to our normal form for constraints we designed a method [16, 21] that simplifies Chan's one while the search space is smaller. Additionally, for each solution, each possibility of the negation is combined with one of the others. All these different solutions are obtained by backtracking.

2.4 Intensional negation and universal quantification

Intensional negation is a novel approach to obtain the program completion by transforming the original program into a new one that introduces the "only if" part of the predicate definitions (i.e., interpreting implications as equivalences). Informally, the *complement* of the terms of the heads of the positive clauses are computed and they are used later as the head of the negated predicate. Given the program (from [1]):

```
even(0).
even(s(s(X))) :- even(X).
```

a new predicate `not__even` is generated that succeeds when `even` fails:

```
not__even(s(0)).
not__even(s(s(X))) :- not__even(X).
```

Even with this informal presentation, it is easy to find two problems with this technique. The first one is related to the presence of new logical variables in the body of a clause. The new program needs to handle some kind of universal quantification construct. Another problem affects the outcomes of the program: while the new program is semantically equivalent to the completed program, the operational behavior can differ. When called with free variables in the goal, the new predicate can generate all the possible values one by one, even when a more general answer can be given. The predicate p defined by the single clause $p(X, X)$. is negated by:

```
not__p(X, Y) :- not__eq(X, Y).
```

assuming that the program only works with natural numbers with `0` and `succ`. The query `not__p(X,Y)`, with an obvious solution $X \neq Y$, will generate infinitely many answers. An answer like $X \neq Y$ can only be replaced by an infinite number of equalities.

We reformulate the transformation by using constraints instead of concrete terms (we assume that disequality constraints are supported using the previously presented technique). Our transformation, when applied to the previous examples, produces the following code:

```
not__even(X) :- X =/= 0, fA(Y, X =/= s(s(Y))).
not__even(s(s(X))) :- not__even(X).
not__p(X, Y) :- X =/= Y.
```

The transformation is fully formalized in [21] and differs from the original one [2] in some significant points. While the original transformation is limited to a restricted class of programs (that models all programs applying a second transformation), our transformation applies to all kinds of programs. Furthermore, it maintains compact outcomes and is designed to produce efficient code. The key point is the use of a single constraint to express the complement of a term, instead of a set of terms.

Let us come back to the universal quantification problem. When a program like this is processed:

```
has_even(L) :- member(X, L), even(X).
```

the transformed program is the following:

```
not__has_even(L) :- forall([X], (not__member(X, L); not__even(X))).
```

The efficient implementation of universally quantified goals is not an easy task. In fact it is an undecidable problem. Our implementation is based on two ideas:

1. A universal quantification of the goal Q over a variable X succeeds when Q succeeds without binding (or constraining) X.
2. A universal quantification of Q over X is true if Q is true for all possible values for the variable X.

It is not possible to generate all possible values (in the presence of a constructor of arity greater than 0) but we can generate all the possible skeletons of values, using new variables. Now, the universal quantification is tested for all this terms, using the new variables in the quantification.

These skeletons are generated incrementally. We start with the simplest one: a variable X. From one skeleton the next one is generated by choosing one term and one variable Y in this term. Given all the possible constructor skeletons (i.e., all program constructors applied to fresh variables) $t_1, \ldots, t_m$ a new skeleton is obtained by replacing the variable Y by each t_i. A Cantor's diagonalization (a method to implement a breadth first strategy of Cartesian products) is used to ensure that all skeletons are generated and checked. For the actual implementation, skeletons do not use variables but new constants that do not appear in the program, i.e., "Skolem constants". Notice that the skeletons grow incrementally, so we only need to check the most recently included terms. The other ones have been checked before and there is no reason to do it again.

As an example, consider a program which uses only natural numbers: the sequence of skeletons for the goal $\forall\ X, Y, Z\ p(X, Y, Z)$ will be the following (where `Sk(i)`, with i a number, represents the ith Skolem constant).

$S_1 = [(Sk(1), Sk(2), Sk(3))]$
$S_2 = [\underline{(0, Sk(1), Sk(2))}, \underline{(s(Sk(1)), Sk(2), Sk(3))}]$
$S_3 = [\underline{(0, 0, Sk(1))}, \underline{(0, s(Sk(1)), Sk(2))}, (s(Sk(1)), Sk(2), Sk(3))]$
$S_4 = [\underline{(0, 0, 0)}, \underline{(0, 0, s(Sk(1)))}, (0, s(Sk(1)), Sk(2)), (s(Sk(1)), Sk(2), Sk(3))]$
$S_5 = [(0, 0, 0), \underline{(0, 0, s(0))}, \underline{(0, 0, s(s(Sk(1))))}, (0, s(Sk(1)), Sk(2)),$
$\quad (s(Sk(1)), Sk(2), Sk(3))]$
$S_6 = \ldots$

In each step, only two elements need to be checked, those that appear underlined. The rest are part of the previous skeleton and they do not need to be checked again.

Universal quantification is implemented by means of the predicate `for_all([X1, ..., Xn], Q, D, S)`, where $X1, \ldots, Xn$ are the universal quantified variables, Q is the goal or predicate, D is the depth to which we want to generate skeletons, and S is an output parameter indicating the success of the evaluation. The initial skeleton $S_1 = [Sk(1), \ldots, Sk(n)]$ of depth 1 is tried. If the goal Q/S_1 is true then the quantification is true. If it fails for a ground case, the quantification is false. Otherwise, the next skeleton of depth 2 is generated and we proceed with it, until a result is obtained or the maximum depth is reached. If the *for_all/4* predicate is not able to achieve a solution at this depth, the predicate informs that it is not possible by binding `S` to `unknown`).

A metapredicate `call_not(`$P(\overline{\mathtt{X}})$`, S)` is used to call the adequate version of `not_`P and return the corresponding result in `S`.

The query evaluation process does not ensure completeness. There are some cases when the generation of skeletons does not find one which is correct or incorrect and the maximum depth is reached. Nevertheless, this solution fails to work properly in very particular cases. Remember that we are not interested in giving the user a universal quantification operator, but just to implement the code coming from the transformation of a negated predicate.

2.5 General constructive negation

Full constructive negation is needed when all the previous techniques are not applicable. While there are several papers treating theoretical aspects of it, we have not found papers dealing with its implementation. The original papers by Chan gave some hints about a possible implementation based on coroutining, but the technique was just sketched. When we have tried to reconstruct it we have found several problems including floundering (in fact it seems to be the reason why constructive negation has been removed from recent Eclipse versions). Thus, we decided to design an implementation from scratch. Up to now, we have achieved only a very simple implementation that certainly needs to be improved. We will sketch the main ideas of the implementation (although this is not the main goal of the paper and we hope to report the details in a forthcoming paper). Recall that we want to use a standard Prolog implementation, so we will avoid implementation-level manipulations.

Full constructive negation can be briefly described in the following terms: In order to compute $\neg Q$ we start an SLD computation for the goal Q. A frontier of Q is a finite set of nodes of the SLD resolution tree such that every resolution branch of Q is either a failure or passes through exactly one node in the set. A frontier can be expressed as $\{(\theta_1, Q_1), \ldots, (\theta_m, Q_m)\}$, where each θ_i is a substitution and Q_i is a subgoal. Any frontier can be interpreted as the logic formula $(\theta_1 \wedge Q_1) \vee \ldots \vee (\theta_m \wedge Q_m)$ (viewing substitutions as equalities) that is equivalent to the original goal Q. To obtain the negation of Q is enough to negate the frontier formula. This is done by negating each component of the disjunction and combining the results. Most of the elements needed for the implementation of the method are also needed for the finite constructive negation case. We already have some code to negate a substitution (that must be reformulated to include predicate calls that can appear in each Q_i), and code to combine the negated solutions.

What is missing is a method to generate the frontier. Up to now we are using the simplest frontier possible: the frontier of depth 1 obtained by doing all possible single steps of SLD resolution. Simple inspection of the applicable clauses can do this. However, we plan to improve it by using abstract interpretation again and detecting the degree of evaluation of a term that the execution will generate.

Using these ideas we have implemented a predicate `cneg` for full constructive negation. Built-in based goals have a special treatment (moving conjunctions into disjunctions, disjunctions into conjunction, eliminating double negations, etc.)

2.6 Implementing negation

Once we have described the main methods implemented together with their limitations, we introduce our most novel proposal: a method for combining these techniques in order to get a correct, complete, and efficient system to handle negation. Our strategy tries to use the simplest possible negation technique for each particular case. Information from global program analysis and some heuristics are used to select among techniques and to optimize the computations involved in the processing of negation. We assume that correct and acceptably accurate analyses are available for the properties of groundness (variables that are bound to a ground term in a certain point of the program), goal delay (identification of delay literals which will not delay, possibly after reordering), and finiteness of the number of solutions.

Our first goal is to produce a (pseudo)predicate `neg` which will compute constructively the negation of any Prolog (sub)goal $\neg G(\overline{X})$, selecting the most appropriate technique. We would also like to generate a specialized version of `neg` for each negated literal in the program (each call to `neg`), using only the simplest technique required. In this process:

1. Groundness of $\overline{X}$ is checked before the call to G. On success simple negation as failure is applied, i.e., it is compiled to `naf(`$G(\overline{X})$`)`.[1]
2. Otherwise, a new program replacing the goal by `(delay(`$\overline{X}$`), naf(`$G(\overline{X})$`))` is generated. Then the "elimination of delays" technique is applied to the new program. If the analysis and the program transformation are able to remove the delay (perhaps moving the goal) the resulting program is used.[2]
3. Otherwise, the finiteness analysis is applied to $G(\overline{X})$. In case of success, then finite constructive negation can be used, transforming the negated goal into `cnegf(`$G(\overline{X})$`)`.
4. Otherwise, the intensional negation approach is tried by generating the negated predicates and replacing the goal by `call_not(`$G(\overline{X})$`, S)`. During this process new negated goals can appear and the same compiler strategy is applied to each of them. If `S` is bound to `success` or `fail` then negation is solved.
5. If everything fails, full constructive negation must be used and the executed goal is `cneg(`$G(\overline{X})$`)`.

The strategy is complete and sound with respect to Kunen 3-valued semantics. This follows from the soundness of the negation techniques, the correctness of the analysis, and the completeness of constructive negation.

The method can be expressed as a Prolog program *scheme* (in the sense that properties of the analyzers and our Prolog predicates are mixed) in the following way:

```
neg(Pred):- ground(Pred),!,naf(Pred).
    % Ground calls. Negation as failure is used.
```

[1]Since floundering is undecidable, the analysis only provides an approximation of the cases where negation as failure can be applied safely. This means that maybe we are avoiding to use the technique even in cases that it could work properly.

[2]Again, the approximation of the analysis could forbid us to apply the methodtar in some cases in which it might still provide a sound result.

```
neg(Pred):- finite(Pred),!,cnegf(Pred).
     % Finite number of solutions. Finite constructive negation.
neg(Pred):- call_not(Pred, S), S==success, !.
     % Intensional negation. Checking of adequate result.
neg(Pred):- cneg(Pred).
     % Full constructive negation.
```

Let us illustrate the behavior of the method by using some simple examples. Consider the following program:

```
 less(0, s(Y)).                         member(X, [X|L]).
 less(s(X), s(Y)) :- less(X, Y).        member(X, [Y|L]) :- member(X, L).

 p1(X) :- member(X, [0, s(0)]),         p3(X) :- neg(less(X, s(s(0)))).
          neg(less(X, s(0))).
 p2(X) :- neg(less(X, s(0))),           p4(X) :- neg(less(s(0), X)).
          member(X, [0, s(0)]).
                                        p5(X) :- neg(less(X, s(X))).
```

Each of the p_i predicates requires a different variant. For p1 the groundness test for variable X succeeds and simple negation as failure can be used, so it behaves as:

```
p1(X) :- member(X, [0, s(0)]), naf(less(X, s(0))).

?- p1(X).
   X = s(0)
```

Applying the "elimination of delays" analysis to program:

```
p2(X) :- (delay(X), naf(less(X, s(0)))), member(X, [0, s(0)]).
```

the delay can be eliminated, reordering the goals as follows:

```
p2(X) :- member(X, [0, s(0)]), naf(less(X, s(0))).

?- p2(X).
   X = s(0)
```

The case for p3 is solved because the finiteness test can be proved to succeed, so the program is rewritten as:

```
p3(X) :- cnegf(less(X, s(s(0)))).

?- p3(X).
   X / 0 ,  X / s(0)
```

p4 needs intensional negation, so the generated program is:

```
not__less(W, Z) :- W =/= 0, fA(X, W =/= s(X)), fA(Y, Z =/= s(Y)).
not__less(s(X), s(Y)) :- not__less(X, Y).
p4(X) :- not__less(s(0), X).

?- p4(X).
   X = 0 ?;
   X = s(0)
```

Finally, p5 needs full constructive negation because the intensional approach is not able to give a result:

```
p5(X) :- cneg(less(X, s(X))).

?- p5(X).
   no
```

3 Evaluating the strategy

3.1 Example programs

As mentioned earlier, one problem that we have faced is the lack of a good collection of benchmarks using negation to be used in the tests. We have however collected a number of examples using negation from logic programming textbooks, research papers, and our own experience teaching Prolog:

- **disjoint**: Simple code to verify that two lists have no common elements. Negation is used to check that elements of the first list are not in the second one.
- **jugs**: There are two jugs, one holding 3 and the other 5 gallons of water, they are both full at the beginning. Jugs can be filled, emptied, and dumped one into the other either until the poured-into jug is full or until the poured-out-of jug is empty. Devise a sequence of actions that will produce 4 gallons of water in the larger jug. Negation is used to check that the status of the jugs is not repeated during the process.
- **robot**: Simulation of the behavior of a robot in an artificial word attending to external perceptions. Negation is used to check that possible new positions for the robot are not dangerous.
- **trie**: Having a list of words and files it finds the list of word-FileList couples that shows the sublist of files, from an initial list, where each word appears. Negation is used when reading words to find the first character that is not alphanumeric.
- **numbers9**: An implementation of balanced tree structures. Negation is used to detect impossible cases.
- **closure**: Transitive closure of a network. Negation is used to avoid infinite loops (detecting repeated nodes). From [19] page 169. (We have studied two implementations of this example.)
- **union**: Union of two lists without repetitions. To check if an element X appears in both lists L_1, L_2 a call to `neg(member (X, L`$_1$`))` is used. From [19] page 154.
- **include**: $include(P, Xs, Ys)$ is true when Ys is the list of the elements of Xs such that $P(X)$ is true. Negation is used to detect elements that do not satisfy the property $P(X)$. From [19] page 227.
- **flatten**: Flattening a list using difference-lists. Negation is used to consider lists that are not empty. From [13] Program 915.2, page 241.
- **lessNodd**: Returns the list of odd natural numbers that are less than a number N. Negation is used to control that a number is not even.
- **friend**: Deduces the relation of friendship between two people using the stored information from a database. Negation is used to restrict the category of friends of a person to those people that are not ancestors or descendants of that person. (We have studied two implementations of this example.)

programs	**const.**	**naf/delay**	**ratio**	**fin.const.**	**ratio**	**intens.**	**ratio**
disjoint1	7440	**780**	9.5	2740	2.7	-	-
disjoint2	3330	-	-	**1120**	2.9	-	-
jugs	8140	**859**	9.4	2175	3.7	<1	x
robot	4600	**1310**	3.5	1900	2.4	-	-
trie	8950	**1850**	4.8	2140	4.1	-	-
numbers9	286779	-	-	-	-	**25230**	11.3
closure1a	5100	**730**	6.9	1450	3.5	140	36.4
closure2a	3520	**560**	6.2	900	3.9	100	35.2
closure3a	10550	**1700**	6.2	2700	3.9	280	37.6
closure1b	26350	D**2240**	11.7	16460	1.6	8570	3.0
closure2b	17400	D**1500**	11.6	10580	1.6	5420	3.2
closure3b	16700	D**4510**	3.7	10120	1.6	16070	1.0
union1	1150	**300**	3.8	320	3.5	189	6.0
union2	20930	-	-	**9470**	2.2	2940	7.1
include1	9020	**1270**	7.1	2680	3.3	170	53.0
include2	9910	-	-	**2995**	3.3	-	-
flatten	32379	**8500**	3.8	12570	2.5	10	x
lessNodd1	58980	**4850**	12.1	17550	3.3	1270	46.4
lessNodd2	7750	**1490**	5.2	2700	2.8	-	-
lessNodd3	>3600000	-	-	-	-	**1540**	x
friend1a	16150	**2280**	7.0	-	-	39500	0.4
friend2a	17630	<**1**	x	-	-	10	x
friend3a	447200	D**4430**	100.9	-	-	43200	10.3
friend4a	>3600000	D**8750**	x	-	-	>3600000	x
friend1b	17350	**3020**	5.74	-	-	9	x
friend2b	17650	<**1**	x	-	-	10	x
friend3b	92500	D**3060**	30.2	-	-	43200	2.1
friend4b	>3600000	D**6050**	x	-	-	171290	x
average			13.0		2.9		18.3

Table 1: Comparing different negation techniques

3.2 Experimental results

We have first measured the execution times in milliseconds for the previous examples when using all the different (applicable) negation techniques that we have discussed, and also noted which technique is selected by our strategy. All measurements were made using Ciao Prolog[3] 1.5 on a Pentium II at 350 Mhz. Small programs were executed a sufficient number of times to obtain repeatable data. The results are shown in Table 1, where the meaning of the different columns is the following:

- **const.** shows the time taken by general constructive negation, i.e., when the negated goal uses `cneg`.

[3]In fact, the negation system is coded as a library module (a Ciao "package"), which includes the corresponding syntactic and semantic extensions (the latter using Ciao's attributed variables). Such extensions apply locally within each module which uses this negation library.

- **naf/delay** uses either `naf` directly or within a delay directive. A 'D' is placed before the time in the second case.
- **fin.const.** is the execution time of the finite version of constructive negation, `cnegf`.
- **intens.** uses the `not_'p'` predicate from the intensional negation program transformation.
- **ratio** columns measure the speedup of the technique to their left w.r.t. constructive negation.

A '–' in a cell means that the technique is not applicable. When the execution time is presented in boldface it indicates that this is the technique selected by our strategy.

It is clear that the technique chosen by our strategy is always equal to or better than general constructive negation. In many cases, it is also the best possible technique. We now study each technique separately:

- Using *naf* instead of general constructive negation results in speed-ups that range from 3.5 to 30.2. The average is more than 8.
- The delay technique, when applicable, has a considerable impact, speeding programs even 100 times.
- The finite version of constructive negation is around 3 times faster than the general version.
- Intensional negation has a more random behavior. Very significant speed-ups are interleaved with more modest results and even some slow-down (*friend1a*).

Probably, the most surprising result is the efficiency of intensional negation. The transformational approach seems the most adequate in those cases provided that we restrict the use of the technique to the case where there are no universal quantifications in the resulting program. On the other hand it is possible that the intensional program may not be able to produce a result (wasting time) and its use is a dynamic decision. Although these problems do not arise often in practice, they are a serious risk. Given the results, we decide to modify the strategy to use intensional negation as the preferable technique, but only when it can be used safely.

As a general conclusion, our strategy clearly produces considerable benefits. It preserves the completeness of general constructive negation but typically at a fraction of the cost.

3.3 Measuring the impact of abstract interpretation

As mentioned previously, the selection strategy and the program optimizations performed make use of information from global program analysis. In our experiments, the information has been obtained and the transformations performed using the analyzers and specializers that are part of the Ciao system's preprocessor, CiaoPP [10, 5].

In particular, from the analysis point of view, the *groundness* analysis has been performed using the domain and algorithms described in [17]. In order to *eliminate* *delays* a technique is used which, given a program with delays, tries to identify those that are not needed, perhaps after some safe reordering of literals, as described in [9, 20]. Finally, in order to determine *finiteness in the number of solutions*, the upper bounds complexity and execution cost analysis has been used [15]: note that an upper bound cost that is not infinity implies a finite number of solutions (an alternative is [3]).

The transformations have been implemented using the specializer in CiaoPP. The source programs always make calls to a version of the generic predicate similar to the `neg` predicate presented in section 2. The specializer creates specialized versions of the generic predicate for each literal calling `neg` in which tests and clauses are eliminated as determined by the information available from the analyzers. For example, if the groundness test is proven true at compile-time, the specializer will automatically eliminate the test and the rest of the clauses of `neg` and eventually even replace the literal calling `neg` with a direct call to `naf`. The nice thing is that this is done automatically by CiaoPP without having to write any additional code.

In order to estimate the advantages obtained by using this approach we now present some experimental results comparing the execution time of the programs that might be generated without the help of the analyzers and the versions produced automatically by the Ciao preprocessor. In the first case, the calls to `neg` always call (a slightly modified version of) the full version of the `neg` predicate. Thus, for example, the groundness test is performed at execution time. The clause to check the finiteness of the goal and then call `cnegf` is removed since such checking cannot be made safely at runtime. Moreover, the delay technique is not used because, in general, it has the risk of floundering. In contrast, the version obtained with the help of the analyzers can remove the groundness check, use the reordering proposed by the elimination of delays, and use the information of the finiteness analysis to call `cnegf`.

Table 2 presents the results. We have also added for reference columns showing the execution time of using `naf` directly and a secure version of `naf`, i.e., checking groundness before. Finally, we have also added the time taken by CiaoPP to perform the analysis and transformation.

The table reveals that the impact of abstract interpretation is significant enough to justify its use. For those examples where `naf` is applicable, the analyzer is able to detect groundness statically in all the cases, so the call to `neg` is replaced by `naf`. It is worth mentioning that the implementation of the dynamic groundness test in Ciao is quite efficient (it is performed at a very low level, inherited from its &-Prolog origins). Even so, the speedup can reach a factor of over 8, and the average is 2.33. The impact of the elimination of delay is even better in general. Notice that if the delay technique is not used, intensional negation could be used instead, which in many cases is a very efficient approach. Even with this drawback, the use of abstract interpretation is helpful. The finiteness analysis avoids usually the use of full constructive negation, and the speed-ups are greater than 3. Notice that the difference between the programs after preprocessing and the direct use of `naf` is irrelevant. The code produced by the preprocessor is better than the secure use of `naf` because of the elimination of groundness tests.

program	with pp.	without pp.	ratio	naf	ratio	secure naf	ratio	prep.
disjoint1	1020	1700	1.66	780	0.76	1469	1.44	78
jugs	969	8419	8.68	859	0.88	1690	1.74	227
robot	1960	3100	1.58	1310	0.66	1800	0.91	700
trie	1890	2450	1.29	1850	0.97	1900	1.00	508
union1	300	350	1.16	230	0.76	300	1.00	119
closure1a	730	2600	3.56	730	1.00	900	1.23	257
closure2a	570	1970	3.45	560	0.98	670	1.17	257
closure3a	1710	5050	2.95	1700	0.99	2010	1.17	257
include1	1099	1180	1.07	1080	0.98	1270	1.15	178
flatten	8859	9300	1.04	8500	0.95	8080	0.91	168
lessNodd1	7310	8670	1.18	4850	0.66	6300	0.86	58
lessNodd2	1780	1830	1.02	1490	0.83	1590	0.89	58
friend1b	3220	3360	1.04	3020	0.93	3180	0.98	198
friend1a	2820	2860	1.01	2280	0.80	2840	1.00	198
average			2.33		0.86		1.10	
closure1b	610	8610	14.11	-	-	-	-	257
closure2b	570	5700	10.00	-	-	-	-	257
closure3b	1800	16300	9.05	-	-	-	-	257
friend3a	3100	43350	13.98	-	-	-	-	198
friend4a	6210	>3600000	x	-	-	-	-	198
friend3b	3100	43400	14.00	-	-	-	-	198
friend4b	6210	171495	27.61	-	-	-	-	198
average			14.79					
disjoint2	1125	3700	3.28	-	-	-	-	78
union2	9590	21010	2.19	-	-	-	-	119
include2	3070	10010	3.26	-	-	-	-	178
average			5.65					
average			2.37		0.86		1.10	

Table 2: Impact of program analysis

References

[1] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. Intensional negation of logic programs. *Lecture notes on Computer Science*, 250:96–110, 1987.

[2] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *JLP*, 8(3):201–228, 1990.

[3] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of Prolog. In *ILPS*, pages 457–471. The MIT Press, 1994.