Intensional Negation of Logic Programs:
examples and implementation techniques

Roberto Barbuti, Paolo Mancarella, Dino Pedreschi, Franco Turini

Dipartimento di Informatica
Università di Pisa
Corso Italia, 40
56100 Pisa - Italy

Abstract

Intensional negation is a transformation technique which, given the Horn clause definitions of a set of predicates p_i, synthesizes the definitions of new predicates $p_i\sim$ the meaning of which is the effective part of the complement of p_i's success set. The main advantage with respect to the standard negation as failure rule is the symmetry in handling both positive and negative information, up the ability of computing non ground negative goals as well as producing non ground output as result of negative queries.

1. Introduction

In the field of deductive data bases [8] and expert systems construction [6, 10, 11] logic programming is gaining momentum. On the other hand the expressiveness of logic programming is still too inadequate for addressing such problems in a natural way. In particular, the ability of handling negative knowledge is a recognized weak point of the logic programming approach.

Up to now, the only kind of negation which has been thoroughly studied is the so called *negation as failure* [7]. Negation as failure is a meta inference rule allowing to prove the negation of a ground goal, when the proof of the corresponding positive goal finitely fails.

The rule has been proved sound and complete for a particular transformation of Horn theories (completed Horn theories) [9]. Furthermore, the soundness and completeness has been proved for certain classes of general logic programs, i.e. programs containing negative literals in the body of clauses [3, 5, 7, 14, 15, 19, 20].

The major drawback of the negation as failure inference rule is that it works correctly only if, during the refutation process, each selected negative literal is ground. In [7, 14, 15, 19, 20] classes of logic theories with the above properties are characterized.

The main contribution of this paper consists in allowing the correct computation of non ground negative goals. As a consequence, the correct answer substitutions for negative goals can be computed. This result is obtained by transforming a Horn logic program (not containing negative literals) into the corresponding *negative* one and querying the latter with the negative goal.

The key idea supporting the transformation is the following. A logical term t (with respect to a first order language L which provides constant and constructor symbols) can be viewed as an intensional representation of the set of all its ground instances. If t does not contain multiple occurrences of the same variable then also its set-theoretic complement can be intensionally represented by means of a finite set of terms [13]. For example, given the usual representation of natural numbers (0, s(0), ...), the complement of the term s(0) can be intensionally represented by the set of terms {0, s(s(x))}. This process can be shown correct provided that all the values of each interpretation domain are obtained by the application of constructors to the constants in L. As an example, the above complement of the term s(0) is correct w.r.t. all the domains satisfying the axiom $\forall x\ (x=0 \lor \exists y\ x=s(y))$. This kind of axiom will be referred to, in the sequel, as *domain closure axiom* (DCA) as in [14]. Actually, intensional negation is based upon the ability of finitely representing the complement of the terms occurring in the clause heads, and this is possible avoiding multiple occurrences of variables in clause heads [13]. This restriction is met by a suitable transformation which turns logic programs into a left-linear form.

The intensional negation approach can be viewed as an extension of the approach presented in [18] in two main respects. First of all, intensional negation is able to deal with logic programs in the general case. Secondly, the ability of computing non ground answer substitutions to negative queries is an attractive feature of intensional negation which, in general, is not met by the approach in [18].

The transformation technique allows a symmetric representation of positive and negative knowledge. Indeed the negative program is a *general* logic program, i.e. negation as failure can be used in clause bodies. In this respect, the transformation technique leads to a computable (by means of a suitable inference rule), intensional representation of the negative knowledge implicit in a logic program. The possibility of handling an intensional representation of negative knowledge is a powerful knowledge engineering method. Indeed, in many applications of deductive data bases and expert systems, the ability of directly checking the implicit negative information can help in debugging and tuning knowledge bases.

2. Proving negative formulæ in logic programming

This section presents some well known results on gathering negative information in logic programming. A thoroughly presentation of this subject can be found in [16]. As pointed out in the introduction, the most important rule for negation is the so called *negation as failure* rule (*naf*) which approximates in an effective way the *closed world assumption* (CWA) [17]. The CWA inference rule states that if a ground atom A is not a logical consequence of a program P then it is possible to infer $\neg$A.

Using the SLD resolution it is possible to prove $\neg$A with respect to a program P under the CWA if the goal $\leftarrow$A has a finitely failed SLD-tree. Obviously, a SLD-tree is not always finitely failed; any proof procedure is semi-decidable and can loop forever. As a consequence, the *naf* rule has been introduced [7], which states that if A is in the SLD finite failure set of a program P, then it is possible to infer $\neg$A with respect to a *completed* program ***P*** (in [16] the notation comp(P) is used instead of ***P***).

According to [16, 2], the SLD finite failure set of P is defined to be the set of all A $\in B_P$ (the Herbrand base of P) for which there exists a finitely failed SLD-tree for P$\cup\{\leftarrow A\}$. The soundness and completeness results for the *naf* rule are given with respect to the completed program ***P*** [9, 7]. The completeness result for the *naf* rule states that if P is a program and A $\in B_P$, then if $\neg$A is a logical consequence of ***P*** then A is in F_P, i.e. the finite failure set of P. This result is independent of the implementation of the proof procedure. Using a fair SLD resolution [16, 12], a sound and complete implementation of F_P is obtained, i.e. the SLD finite failure set is equal to F_P.

3. Intensional negation

In this section the approach to the synthesis of negative programs is presented by means of examples of increasing complexity. The approach is based on the *completion* of a logic program which, roughly speaking, corresponds to replace the if connectives ($\leftarrow$) in the program with iff connectives ($\Leftrightarrow$), and the domain closure axiom.

In the sequel, programs with no multiple occurrences of the same variable in clause heads are termed *left-linear*, and a variable occurring in a clause body but not in the corresponding clause head is termed *right-free*. The following subsections present the application of intensional negation to left-linear programs without right-free variables, to left-linear programs with right-free variables and to non left-linear programs respectively.

3.1. Left-linear programs without right-free variables

The following program Even defines the set of even numbers, represented as terms of the Herbrand universe {0, s(0), s(s(0)), . . . }.

even(0) $\leftarrow$
even(s(s(x))) $\leftarrow$ *even*(x)

The completed program ***Even*** corresponding to Even is

Even

$\forall x \ (even(x) \Leftrightarrow x=0 \lor \exists y\ (x=s(s(y)), even(y)))$

We are omitting here the axiomatization of the equality theory on the Herbrand universe, defining predicate =, and the associated DCA [14]. Subformulæ such as $\exists y\ x=s(s(y))$ will be referred in the sequel as *positive guards* (for variable x).

The aim is to synthetize the definition of a new predicate, *even~*, such that *even~*(n) holds iff *even*(n) is provably false or, in other words, the theory ***Even*** $\cup\{even(n)\}$ has no models. The flavor of the transformation can be obtained by the examination of its main steps.

1) Applying logical negation to the axiom for *even* and interpreting occurrences of negative atoms $\neg even(t)$ as occurrences of the positive atom *even~*(t), we obtain the axiom

$\forall x\ (even\text{\~{}}(x) \Leftrightarrow \neg\ (x=0), \neg\ \exists y\ (x=s(s(y)), even(y)))$

which can be further transformed into

$\forall x\ (even\text{\~{}}(x) \Leftrightarrow x\neq 0, \forall y\ (x\neq s(s(y)) \lor even\text{\~{}}(y)))$

2) Let x be an arbitrary term, then the subformula $\forall y\ (x\neq s(s(y)) \lor even\text{\~{}}(y))$ can be shown equivalent to

$\forall y\ (x\neq s(s(y))) \lor \exists y\ (x=s(s(y)), even\text{\~{}}(y))$.

Hence, by substituting $\forall y\ (x\neq s(s(y)) \lor even\text{\~{}}(y))$ with this last formula and by $\land$-distribution we obtain

$$\begin{aligned}\forall x\ (even\text{\~{}}(x) \Leftrightarrow\ & x\neq 0, \forall y\ x\neq s(s(y)) \quad \lor \\ & x\neq 0, \exists y\ (x=s(s(y)), even\text{\~{}}(y))\ ).\end{aligned}$$

Subformulæ such as $\forall y\ x\neq s(s(y))$ are termed *negative guards* (for variable x).

3) The next step consists in transforming negative guards into (a disjunction of) positive guards, still preserving their logical meaning, i.e. computing the intensional complement of the term involved in the right hand side of a negative guard. The technique has been introduced for the negation of predicates in the framework of symbolic evaluation [1]. Here the technique is implemented by a transformation, named *Ex*, which will be discussed in section 4. The application of *Ex* to the negative guards in the example yields:

- $Ex(x\neq 0) \equiv \exists y\ x=s(y)$

- $Ex(\forall y\ x\neq s(s(y))) \equiv x=0 \lor \exists z\ (x=s(z), Ex(\forall v\ z\neq s(v))) \equiv x=0 \lor \exists z\ (x=s(z), z=0)$
  $\equiv x=0 \lor x=s(0)$

The definition of *even~* becomes:

$\forall x\ (even\sim(x) \Leftrightarrow\ \exists y\ x=s(y), x=0\ \vee$
$\qquad \exists y\ x=s(y), x=s(0))\ \vee$
$\qquad \exists z\ x=s(z), \exists y\ (x=s(s(y)), even\sim(y))\ ).$

4) Conjunctions of formulæ with positive guards for the same variable can be turned into a single formula by finding the *most general unifier*, if any, of their guards, and propagating this m.g.u. to the remaining parts. In the example

• $\exists y\ x=s(y), x=0\ \Rightarrow\ \mathbf{ff}$	Since the m.g.u. of s(y) and 0 does not exist
• $\exists y\ x=s(y), x=s(0))\ \Rightarrow\ x=s(0)$	m.g.u.(s(y), s(0)) = {<y,0>}
• $\exists y\ x=s(y), \exists z\ x=s(s(z))\ \Rightarrow\ \exists z\ x=s(s(z))$	m.g.u.(s(y),s(s(z))))={<y,s(z)>}

Hence one obtains the completed definition of *even~*
$\forall x\ (even\sim(x) \Leftrightarrow\ x=s(0)\ \vee\ \exists y\ (x=s(s(y)), even\sim(y))\ ).$

5) Finally, the logic program for the new predicate is

even~(s(0)) ←
even~(s(s(x))) ← *even~*(x)

which, as one might expect, defines exactly the set of odd numbers.

Querying such a program with a ground goal such as ← *even~*(s(0)) yields the same answer as querying the original program with a goal ← ***naf*** *even*(s(0)), where ***naf*** stands for the *negation as failure* operator. Indeed, such a goal is nothing but a test (which does not involve computing answer substitutions) and this is the only kind of goal that the query evaluation process based on the *naf* rule is able to handle correctly. More properly, the evaluation of a non ground goal such as ← ***naf*** *even*(x) is simply a way to prove whether $\forall x\ \neg even(x)$ holds or not w.r.t. to the completed program. On the other hand, the goal ← *even~*(x) using intensional negation is a way to prove whether $\exists x\ \neg even(x)$ holds or not and, as usual, to compute the values for x which make the formula true. In the example, the goal ← ***naf*** *even*(x) fails while the goal ← *even~*(x) succeeds enumerating the odd numbers as solutions. This provides a nice notion of symmetry between the use of positive and negative knowledge.

Consider, as a further example, the following program LessOrEqual wich defines the relation "≤" on natural numbers.

LessOrEqual(0, x) ←
LessOrEqual(s(x), s(y)) ← *LessOrEqual*(x, y).

The completed definition of *LessOrEqual* is

$$\forall x\ (\mathit{LessOrEqual}(x_1, x_2) \Leftrightarrow \quad x_1{=}0 \ \vee$$
$$\exists y_1 y_2\ (x_1{=}s(y_1),\ x_2{=}s(y_2),\ \mathit{LessOrEqual}(y_1, y_2))\ )$$

The next steps of the transformation yield the following results:

$$\forall x\ (\mathit{LessOrEqual}{\sim}(x_1, x_2) \Leftrightarrow$$
$$x_1{\neq}0,\ (\forall y_1\ x_1{\neq}s(y_1) \vee \forall y_2\ x_2{\neq}s(y_2)) \quad \vee$$
$$x_1{\neq}0,\ \exists y_1 y_2\ (x_1{=}s(y_1),\ x_2{=}s(y_2),\ \mathit{LessOrEqual}{\sim}(y_1, y_2))\ )$$

$$\forall x\ (\mathit{LessOrEqual}{\sim}(x_1, x_2) \Leftrightarrow$$
$$\exists y_1\ x_1{=}s(y_1),\ x_2{=}0 \quad \vee$$
$$\exists y_1 y_2\ (x_1{=}s(y_1),\ x_2{=}s(y_2),\ \mathit{LessOrEqual}{\sim}(y_1, y_2))\ )$$

The logic program corresponding to the above definition is

LessOrEqual~(s(x), 0) ←
LessOrEqual~(s(x), s(y)) ← *LessOrEqual~*(x, y)

which is the expected definition of the relation ">".

As mentioned before, intensional negation allows querying a logic program with arbitrary goals containing also negative atoms. As an example, the query

← *LessOrEqual~*(x, s(s(0)))   (•)

gives the only answer x=s(s(s(y))) which denotes the set of numbers greater than 2, while the corresponding query using the *naf* rule

← ***naf*** *LessOrEqual*(x, s(s(0)))   (••)

fails. One might wonder that the behavior of (•) can be simulated using the original program extended with the clauses

nat(0) ←
nat(s(x)) ← *nat*(x)

and rephrasing (••) as

← *nat*(x), ***naf*** *LessOrEqual*(x, s(s(0)))   (•••)

In this way *nat* acts as a generator for numbers and the negative atom is selected with ground terms bound to x. Nevertheless, this query is somewhat weaker than (•), since the evaluation of (•) gives the whole solution in one shot, whereas (•••) diverges enumerating all the ground solutions. As this example shows, the symmetry between the treatment of positive and negative knowledge is pervasive up to the ability of producing *non ground output.*

3.2. Left-linear programs with right-free variables

The sample programs which have been considered so far do not contain *right-free* variables, i.e. variables occurring only in the body of a clause. Consider for instance the program

$$p(x) \leftarrow q(x,z)$$
$$q(a,b) \leftarrow$$

Applying the same transformation steps of the previous examples, one would obtain the following intensional negation for predicates p and q

$$p\sim(x) \leftarrow q\sim(x,z) \qquad (*)$$
$$q\sim(b,x) \quad \leftarrow$$
$$q\sim(a,a) \quad \leftarrow$$

Then, both p(a) and p~(a) are provable in the whole theory, i.e. the theory is inconsistent since p~ is viewed as an effective counterpart of $\neg$p. The problem is that the right-free variables (such as z in the clause of p) are *unguarded* existentially quantified variables in the completed definition and thus they become unguarded *universally* quantified variables in the negative program. The completed definition of p is

$$\forall x \quad (p(x) \Leftrightarrow \exists z\, q(x,z))$$

hence

$$\forall x \quad (\neg p(x) \Leftrightarrow \forall z\, \neg q(x,z))$$

while the actual meaning of (*) is

$$\forall x \quad (p\sim(x) \Leftrightarrow \exists z\, q\sim(x,z)).$$

Although it is possible to replace negative guards (i.e. universally quantified formulæ involving predicate =) with some appropriate existentially quantified construction, this is not the case when arbitrary predicates are involved, as in $\forall z\, \neg q(x,z)$. However, for any substitution of a term t for x, if the proof of q(t,z) finitely fails (resp. succeeds) we obtain a proof of $\forall z\, \neg q(t,z)$ (resp. $\neg\forall z\, \neg q(t,z)$) [9]. This means that the *naf* rule is appropriate to compute formulæ of such kind, since there is no need to gather output substitutions for the free variables as z in the example. Thus, in the program obtained by intensional negation, a formula like $\forall z\, \neg q(x,z)$ can be implemented by

$$q\sim(x,w),\ \textit{\textbf{naf}}\ q(x,z) \qquad (\bullet)$$

where ***naf*** q(x,z) is called a *naf literal.* The evaluation of ***naf*** q(x, z) succeeds if and only if the goal $\leftarrow$ q(x,z) finitely fails. From a computational point of view, the evaluation of q~(x,w) in (•) provides candidate substitutions $t_1, t_2,\dots$ for x, such that $\exists w\, \neg q(t_i,w)$ holds, and the computation of ***naf*** $q(t_i,z)$ filters those substitutions t_i such that $\forall z\, \neg q(t_i,z)$ holds too. It is worth noting that the use of different variable names for right-free variables avoids undesired interferences between the computations of the two subformulæ. The appropriate refutation procedure (SLDIN resolution [4]) for programs with intensional negation has to cope with non ground *naf* literals which, as shown above, act as filters for correct solutions. To accomplish this task, the SLDIN refutation procedure uses a computation rule which selects a *naf* literal if the whole goal is composed of *naf* literals only, and applies the negation as failure rule to *naf* literals. Thus, SLDIN guarantees that *naf* literals are selected after the non-naf literals have provided the candidate solutions.

It is worth noting that SLDIN resolution deals with queries which are *non allowed queries* for SLDNF resolution [16]. Nevertheless, when a *naf* literal is selected the SLDIN resolution never *flounders* [15] since the negation as failure rule is actually used only to prove a universally quantified theorem.

Concluding the example, the intensional negation is

p~(x) ← q~(x,w), ***naf*** q(x,z)	p(x) ← q(x,w)
q~(b,x) ←	q(a,b) ←
q~(a,a) ←	

Again, a non ground query such as ← p~(x) succeeds with the answer x=b while ←***naf*** p(x) fails. Moreover the inconsistency pointed out above does not hold anymore since ← p~(a) fails under SLDIN resolution.

In the case of clauses with right-free variables occurring in more than one literal in its body, as in

p(x,y) ← q(x,w), r(w,y)

intensional negation is carried out in two steps. First, the clause is replaced by an equivalent one with only one literal in its body, introducing a new predicate symbol. In the example

p(x,y) ← s(x,y,w)
s(x,y,w) ← q(x,w), r(w,y).

Second, intensional negation of the transformed program is computed as in the previous example and a *naf* literal is introduced for each definition with right-free variables. In the example

p~(x,y) ← s~(x,y,w), ***naf*** s(x,y,z)
s~(x,y,w) ← q~(x,w)
s~(x,y,w) ← r~(w,y).

Whenever a clause contains only right-free variables, as in

p(a) ← q(x)

the definitions computed by intensional negation only contain *naf* literals. In the example this leads to the definition

p~(a) ← ***naf*** q(x).

Notice that in the body of the clause defining p~(a) there is no need to introduce the conjunction q~(x), ***naf*** q(y) since no candidate solution has to be computed. In fact, from the completed definition of p we obtain

$p(a) \Leftrightarrow \exists y\, q(y)$ i.e. $\neg p(a) \Leftrightarrow \forall y\, \neg q(y)$.

Thus, in order to prove p~(a) is sufficient to test whether $\forall y \neg q(y)$ holds or not, which is just accomplished by means of the *naf* literal ***naf*** q(y).

As a final example, consider the intensional negation of a program for the ancestor relation, where *Parent*(x,y) means y is a parent of x, *Ancestor*(x,y) means y is an ancestor of x.

Parent(John, Mary) ←
Parent(John, Bill) ←
Parent(Mary, Paul) ←
Parent(Bill, Anne) ←
Ancestor (x,y) ← *Parent*(x,y)
Ancestor (x,y) ← *Ancestor*(x,z), *Parent*(z,y).

First of all, since the last clause contains right-free variables in more than one literal, it is replaced by a new clause which makes use of a new predicate symbol, say *ProperAncestor*. This transformation yields the following program

Parent(John, Mary) ←
Parent(John, Bill) ←
Parent(Mary, Paul) ←
Parent(Bill, Anne) ←
Ancestor (x,y) ← *Parent*(x,y)
Ancestor (x,y) ← *ProperAncestor*(x,y,z)
ProperAncestor(x,y,z) ←*Ancestor*(x,z), *Parent*(z,y).

The definitions computed by intensional negation are

Parent~(John, John) ←
Parent~(John, Paul) ←
.
.
.
Parent~(Paul, Mary) ←
Ancestor~(x,y) ← *Parent~*(x,y), *ProperAncestor~*(x,y,z), ***naf*** *ProperAncestor*(x,y,w)
ProperAncestor~(x,y,z) ← *Ancestor~*(x,z)
ProperAncestor~(x,y,z) ← *Parent~*(z,y)

Notice that the clause defining *Ancestor~* states that y is not an ancestor of x if neither y is a parent of x nor y is a proper ancestor of x through any z. Under SLDIN-resolution the query ←*Ancestor~*(Mary,y) succeeds enumerating the solutions Mary, Bill, John, Anne.

3.3. Non left-linear programs

A final issue concerns intensional negation of *non left-linear* programs, i.e. programs with clauses containing multiple occurrences of the same variable in their heads. Consider the following program which states the relation between a number and its successor:

$plus1(x, s(x)) \leftarrow$

The first step is to apply a *linearization algorithm* which turns each program into an equivalent left-linear form, thus turning back to the previous cases. The linearization of *plus1* is:

$plus1(x,s(y)) \quad \leftarrow \quad eq(x,y)$

where *eq* is the equality predicate defined by the clause

$eq(x, x) \quad \leftarrow.$

Furthermore, the completion of *plus1* over which intensional negation acts is:

$$\forall x_1\, x_2\ (plus1(x_1, x_2) \Leftrightarrow \exists y\ (x_2{=}s(y),\ eq(x_1, y)) \qquad (1)$$

Notice that we have distinguished = from *eq*: The former is introduced when the completion is carried out, while the latter is introduced by the linearization algorithm in order to turn the program into a left-linear form. From a semantic point of view both = and *eq* are the identity relation over the Herbrand universe, but they are handled differently by intensional negation. Consider the intensional negation of (1):

$$\forall x_1\, x_2\ (plus1{\sim}(x_1, x_2) \Leftrightarrow \forall y\ (x_2{=}s(y) \vee eq{\sim}(x_1, y))$$

As before, we distinguish between ≠ and *eq~*: In particular ≠ appears only in negative guards which can be further transformed into a disjunction of positive guards, while *eq~* is handled like an ordinary predicate symbol. In the case of our running example:

$$\forall x_1\, x_2\ (plus1{\sim}(x_1, x_2) \Leftrightarrow\ x_2{=}0 \vee\ \exists y\ \ (x_2{=}s(y) \vee eq{\sim}(x_1, y))$$

Putting the above definition into its Horn clause form one gets:

$plus1{\sim}(x_1, 0) \quad \leftarrow$

$plus1{\sim}(x_1, s(y)) \quad \leftarrow eq{\sim}(x_1, y)$

which defines exactly the set of pairs of naturals <n, m> such that m≠n+1, provided that *eq~* defines the set of pairs <n, m> such that m≠n. This is achieved providing the following *ad hoc* intensional negation of the predicate *eq*:

$eq{\sim}(0, s(x)) \quad \leftarrow$

$eq{\sim}(s(x), 0) \quad \leftarrow$

$eq{\sim}(s(x), s(y)) \quad \leftarrow eq{\sim}(x, y).$

Indeed, = and *eq* make explicit two different roles played by unification in logic programming. In fact the latter makes explicit the use of unification to state constraints of the kind *the same (unspecified) object must occur in different positions within the terms of a clause head.* In the previous example, $plus(x, s(x)) \leftarrow$ means that the relation *plus1* holds between a generic natural number n and the natural number obtained applying the constructor s to n itself.

Now, consider the definition $p(x, x) \leftarrow$, which means that each object is related via p with itself and with no other object. Answering the question *which objects are not related via* p*?* one can just say *objects which are not identical.* Hence, the definition $p{\sim}(x, y) \leftarrow$ is incorrect, since it would imply also that an object is *not* related via p with itself, and this is unconsistent with respect to the definition of p. A correct definition of p~ can be achieved using the *eq~* predicate: p~(x, y) ← *eq~*(x, y), which means *if objects* x *and* y *are not identical, then they are not related via* p.

On the other hand, the = predicate makes explicit the use of unification to describe the structure that objects should exhibit in order to be candidate components of a tuple satisfying some relation. This is what is done by positive guards. Take for instance the definition p(s(x)) ← q(x) , which means *a number is in the set denoted by* p *if* i) *it is the successor of some other number* n, *and* ii) n *satisfies* q. With the *if and only if* interpretation one can safely say that a number which is not the successor of any other number is in the complement of p. This is exactly the meaning of the definition p~(x) ← ∀y x≠s(y) i.e. p~(0) ← . Furthermore one has to state that a number is in the complement of p if it satisfies the condition i) above but it does not satisfiy the condition ii). This is exactly the meaning of the definition p~(s(x)) ← q~(x). The use of predicate = makes explicit the role of conditions analogous to condition i) above.

It is important to notice that the predicate *eq~* is not computationally equivalent to, say, *non-unification* : for instance, the unification of the terms x and s(x) fails because of the occur check, while the refutation of ← *eq~*(x, s(x)) diverges, even if any ground instance of such a goal succeeds. This suggests that intensional negation is not equivalent to non-unification: The results in [4] characterize the behavior of intensional negation in terms of a suitable theory. But the use of predicate *eq* (and hence *eq~*) guarantees that all programs that are to be intensionally negated can be brought into a left-linear form, and this makes possible to translate systematically negative guards into finite disjunctions of positive guards [13]. Looking at the clausal program resulting from intensional negation, it is worth noting that guards (involving predicate =) are absorbed again into unification, while the occurrences of predicate *eq~* are not, since *eq* is handled like any other predicate symbol. As a consequence, intensional negation yields always left-linear programs, apart from the predicate *eq* which is handled in a special way.

4. Computing transformations of negative guards

A central issue in the intensional negation approach is the transformation of negative guards into their existential form. As mentioned in the introduction, a (positive or negative) guard can be viewed as an intensional definition of a subset of the Herbrand universe [1]. For instance, referring to the Herbrand universe of naturals, the positive guard ∃y x=s(y) denotes the set {x | x≥1}, while the negative guard ∀y x≠s(s(y)) denotes the set {0,1}. Under *DCA*, a set S described by a negative guard can always be described by a disjunction of positive guards, corresponding to an intensional constructive definition of S, since each universally quantified variable occurs exactly once in a negative guard [13]. Thus, a transformation *Ex* which, given a negative guard ∀y x≠t builds the equivalent disjunction of positive guards, can be defined by structural induction on the term t. The general definition of *Ex* can be found in [4], while an instance of *Ex* is shown below with respect to the Herbrand universe of naturals.

(1) $Ex\ (\forall y\ x \neq y) \equiv \mathbf{ff}$

(2) $Ex\ (x \neq 0) \equiv \exists y\ x=s(y)$

(3) $Ex\ (\forall y\ x \neq s(t)) \equiv x=0 \lor \exists z\ x=s(z), Ex\ (\forall y\ z \neq t)$.

For instance,

$$Ex\ (\forall y\ x \neq s(s(y))) \equiv x=0 \lor x=s(0)$$

$$Ex\ (x \neq s(0)) \equiv x=0 \lor \exists z\ x=s(z), Ex\ (z \neq 0) \equiv x=0 \lor \exists z\ x=s(z), \exists y\ z=s(y) \equiv$$
$$x=0 \lor \exists y\ x=s(s(y))$$

The last step uses the obvious reduction

$$\exists z\ x=t, \exists y\ z=t' \equiv \exists y\ x=t\ [t'/z]$$

A simple way to implement *Ex* is achieved using the Horn clause definition of the inequality predicate *eq~* and querying it with appropriate goals. This technique is shown by the following example. Consider the Herbrand universe over constants a and b and the binary constructor f. The predicate*eq~* is defined by

eq~ (a,b) ←
eq~ (a, f(x,y)) ←
eq~ (b,a) ←
eq~ (b, f(x,y)) ←
eq~ (f(x,y), a) ←
eq~ (f(x,y), b) ←
eq~ (f(x,y), f(v,w)) ← *eq~* (x,v)
eq~ (f(x,y), f(v,w)) ← *eq~* (y,w).

Next, consider the negative guard $\forall y\ x \neq f(a,y)$. The solutions for x satisfying this guard can be obtained querying the above program with the goal ← *eq~* (x, f(a,*any*)) where *any* is a new constant symbol which acts as a *universal Skolem constant.* These solutions are x=a, x=b, x=f(b,y), x=f(f(y,z),w). The use of the constant *any* forces the failure of recursive calls like *eq~* (z,*any*), as in case (1) of the definition of *Ex*. Interpreting a solution like x=f(b,y) as $\exists y\ x=f(b,y)$ and taking the disjunction of all the solutions we obtain the formula $(x=a \lor x=b \lor \exists y\ x=f(b,y) \lor \exists yzw\ x=f(f(y,z),w))$ which is exactly $Ex\ (\forall y\ x \neq f(a,y))$.

The use of this implementation technique is twofold. Of course, it can be viewed as an implementation of the *Ex* transformation, as mentioned above. On the other hand, it can be used as the basis for a slight variant of intensional negation which directly replaces negative guards with appropriate calls of predicate *eq~* , possibly involving the *any* constant.

Referring back to the predicate *even~* of sec. 3.1., after step 2) of the transformation, i.e.

$$\forall x\,(\textit{even}\sim(x) \Leftrightarrow \begin{array}{l} x\neq 0,\ \forall y\ x\neq s(s(y)) \quad \vee \\ x\neq 0,\ \exists y\ (x=s(s(y)),\textit{even}\sim(y))\ ) \end{array}$$

the guards can be replaced by appropriate calls to predicates *eq* and *eq~* as follows

$$\forall x\,(\textit{even}\sim(x) \Leftrightarrow \begin{array}{l} \textit{eq}\sim(x,0),\ \textit{eq}\sim(x,\ s(s(\textit{any}))) \quad \vee \\ \textit{eq}\sim(x,0),\ \textit{eq}\ (x,s(s(y))),\textit{even}\sim(y))\ ). \end{array}$$

At last, the clausal definition of *even~* is

$$\textit{even}\sim(x) \leftarrow \textit{eq}\sim(x,0),\ \textit{eq}\sim(x,\ s(s(\textit{any})))$$
$$\textit{even}\sim(x) \leftarrow \textit{eq}\sim(x,0),\ \textit{eq}\ (x,\ s(s(y))),\ \textit{even}\sim(y).$$

Finally, notice that this implementation technique is particularly useful in some application area of logic programming, such as deductive databases and expert systems, where the knowledge domain is likely to change dynamically. In such cases, using the standard form of intensional negation one is compelled to recompute the negative predicates whenever the knowledge domain changes, since the clauses for negative predicates embed the transformation of negative guards. On the other hand, the explicit use of predicates *eq* and *eq~* avoids this overall recomputation, since the redefinition of *eq* and *eq~* is only needed.

5. Conclusions

In [4] the formalization of intensional negation is carried out. The main results concern the soundness and completeness of SLDIN-resolution. The soundness theorem states that if the proof of $p\sim(t)$ succeeds under SLDIN-resolution with answer substitution λ, then $\neg p(t\,\lambda)$ can be inferred under the negation as failure rule. This result implicitly gives the semantic equivalence between intensional negation and the negation as failure rule. On the other hand, if λ is a substitution such that $\neg p(t\lambda)$ can be inferred under the negation as failure rule, than the proof of $p\sim(t)$ under SLDIN-resolution succeeds with a (possibly infinite) collection of answer substitutions $\{\lambda_i\}$ such that each instance of $t\lambda$ can be obtained as an instance of $t\lambda_i$, for some i. If only left-linear programs are taken into account, this completeness result can be strenghtened, in the sense that the finiteness of the collection $\{\lambda_i\}$ is guaranteed. Roughly speaking, if non left-linear programs are the matter of concern, intensional negation makes use explicitly of the predicate *eq~* which in some cases gives an infinite set of solutions. As an example, consider the program

$$p(x,\ x) \leftarrow$$

defined over natural numbers. In order to compute its intensional negation, the clause is transformed into its semantically equivalent left-linear form

p(x,y) ← eq(x,y)

and than the computed definition for $p\sim$ is

$p\sim$(x,y) ← $eq\sim$(x,y).

As shown in a previous example, the definition of $eq\sim$ in this case is

$eq\sim$(0, s(x)) ←

$eq\sim$(s(x), 0) ←

$eq\sim$(s(x), s(y)) ← $eq\sim$(x, y).

Now, the proof of ← p(x,s(x)) finitely fails under standard SLD-resolution because of *occur checking*. On the other hand, the proof of ← $p\sim$(x,s(x)) under SLDIN-resolution diverges with answer substitutions x=0, x=s(0), x=s(s(0)),... since these are the substitutions computed by the derived goal ← $eq\sim$(x, s(x)).

The work described in the paper is only a first step of a larger and more ambitious research effort. The general aim of this effort could be summarized by the motto *putting logic theories together as a knowledge engineering tool.* This means providing a number of operators on logic theories capable, for example, of joining them, negating them and so forth. Such a set of operators along with constructs for composing them would provide a sound, formal tool for manipulating chunks of knowledge represented as logic programs.

The intensional negation we have proposed is only a small part of this overall construction. Indeed, the most needed extension is the possibility of writing general programs, in the sense of using the negation of the paper freely in building logic programs. This further extension, besides providing a fully usable operator for manipulating theories (for example with such an extension p~~≡ p can be proved) has some advantages *per se*. Indeed, as discussed in the paper, the evaluation of a goal p~(t) where t is a non-ground term leads to the computation of all the instances t' of t such that the proof of p(t') finitely fails. This possibility seems a notable extension to the expressive power of logic programming in that it allows queries of the form *all elements for which a certain property does not hold.* The extension seems to be of importance especially in deductive data bases applications and expert systems where the negative knowledge is essentially finite. Work done so far in this direction seems to point out that the computation rule SLDIN extends smoothly to the case of general programs.

Another drawback of intensional negation seems to be the inefficiency of the resulting programs. An open problem is the study of optimization techniques able to improve this situation.

References

[1] Ambriola,V., Giannotti,F., Pedreschi,D., Turini,F. "Symbolic Semantics and Program Reduction". **IEEE Trans. on Soft. Eng.** SE-11,8 (Aug.85) 784-794.

[2] Apt,K.R., Van Emden, M.H. "Contribution to the Theory of Logic Programming". **J.ACM**, 29, 3, (1982) 841-862.

[3] Aquilano,C., Barbuti,R., Bocchetti,P., Martelli,M. "Negation as Failure: Completeness of the Query Evaluation Process for Horn Clause Programs with Recursive Definitions".**Journal of Automated Reasoning**, 2 (1986) 155-170.

[4] Barbuti,R., Mancarella,P., Pedreschi,D., Turini,F. "Intensional Negation of Logic Programs". submitted for publication to the **Journal of Logic Programming** (1986).

[5] Barbuti,R., Martelli,M. "Completeness of the SLDNF-Resolution for a Class of Logic Programs". Proc. of 3rd Int. Conf. on Logic Programming, London, July 14-18 (1986).

[6] Barr,A., Feigenbaum,E.A. (Eds.) **The Handbook of Artificial Intelligence**. Vol.1, Pitman, London (1981).

[7] Clark,K.L. "Negation as Failure". in **Logic and Data Bases** (Gallaire, H. and Minker,J. Eds.) Plenum, New York (1978) 293-322.

[8] Gallaire,H., Minker,J., Nicolas,J.M. "Logic and Databases: a Deductive Approach". **ACM Comp.Surv.** 16,2 (June 1984) 153-186.

[9] Jaffar,J., Lassez,J.-L., Lloyd,J.W. "Completeness of the Negation-as-Failure Rule". Proc. 8th Int. Joint Conf. on Art. Int., Karlsruhe (1983) 500-506.

[10] Kowalski, R.A. **Logic for Problem Solving**. Elsevier North Holland, New York (1979).

[11] Kowalski, R.A. "Logic Programming". Proc. IFIP 83, Paris (1983) 133-145.

[12] Lassez,J.-L., Maher,M. "Closures and Fairness in the Semantics of Logic Programming". **TCS**, 29 (1984)

[13] Lassez,J.-L., Marriot,K. "Explicit and Implicit Representation of Terms Defined by Counter Examples". to appear in **Journal of Automated Reasoning.**

[14] Lloyd,J.W., Topor,R.W. "A Basis for Deductive Data Base Systems". **Journal of Logic Programming**, 2,2 (1985) 93-103.

[15] Lloyd,J.W., Topor,R.W. "A Basis for Deductive Data Base Systems II". **Journal of Logic Programming**, 1 (1986) 55-67.

[16] Lloyd,J.W. **Foundations of Logic Programming**. Springer Symbolic Computation Series, Berlin (1984).

[17] Reiter,R. "On Closed World Data Bases". in **Logic and Data Bases** (Gallaire, H. and Minker,J. Eds.) Plenum, New York (1978) 55-76.

[18] Sato,T., Tamaki,H. "Transformational Logic Program Synthesis". Proc. Conf. on Fifth Generation Computer Systems, (1984).

[19] Shepherdson,J.C. "Negation as Failure: a Comparison of Clark's Completed Data Bases and Reiter's Closed World Assumption". **Journal of Logic Programming**, 1,1 (1984) 51-79.

[20] Shepherdson,J.C. "Negation as Failure II". **Journal of Logic Programming**, 2,3 (1985) 185-202.