Negative Subgoals with Free Variables

Matthew L. Ginsberg

Computer Science Department
Stanford University
Stanford, California 94305

Negative Subgoals with Free Variables

Abstract

PROLOG systems currently work with the restriction that negative subgoals not contain free variables, although some authors have suggested that this restriction can be lifted by running the theorem prover to exhaustion on the subgoal being considered. We suggest that this approach is likely not to be computationally viable in practice, and describe and formalize a new approach to dealing with quantified default queries that avoids this difficulty.

1 Introduction

Consider the following PROLOG fragment:[1]

```
f :- not(g(x)).
g(A).
```

The interpretation we will assign to the implication `f :- not(g(x))` will be that it is universally quantified over x, and therefore corresponds to the implication

$$\forall x.[\neg g(x) \supset f] \tag{1}$$

or, equivalently,

$$[\exists x.\neg g(x)] \supset f.$$

Given this interpretation, what should the response be if we provide the system with the query `f`?[2]

Intuitively, we would expect the query to succeed, since f follows from any negative instance of g, and the only known positive instance of g is for the argument A. In practice, however, this is not the case. The standard evaluation procedure used by PROLOG will fail for f, since the negative subgoal $\neg g(x)$ succeeds (by binding x to A).

[1]Throughout this paper, we will reverse the PROLOG convention of capitalizing variables and leaving object constants in lower case; the reason for this is to establish uniformity with the mathematical convention of having variables in lower case. We will also take letters late in the alphabet (x, y, z) to be variables, and letters early in the alphabet (A, B, C) to be object constants.

[2]Conventional PROLOG syntax [6] interprets `f :- not(g(x))` by passing the quantifier through the implication, effectively obtaining $[\forall x.\neg g(x)] \supset f$ instead of (1). By doing this, some of the problems we will be discussing are avoided, although rules such as `f(x) :- not(g(x))` remain problematic if the variable `x` is not bound in the query `f(x)`. There is a growing consensus that (1) is a more natural interpretation in any event [1, 5, 18].

This conclusion is also supported by the work by Gelfond and Lifschitz [10], which augments a PROLOG program with domain closure and uniqueness of names assumptions. Thus in the above example, they would include an axiom stating that A was the only object in the domain in question. The query f would then fail as a result.

The counterintuitive nature of this conclusion is apparent if we modify our program to become the following:

```
f :- not(g(x)).
g(A).
h(B).
```

Now Gelfond and Lifschitz would have the query f succeed, since there is a new object, B, appearing in the domain description. This seems unjustified, however, since the information about B has nothing to do with f or with g.[3]

The problems become more subtle if we allow the potential proof of g to itself be an argument using negation as failure:

```
f :- not(g(x)).
g(x) :- not(h(x)).
h(A).
```

Intuitively, f is true except if g holds for all values of its argument; g holds except where h does. Since h is true at A but not elsewhere, g is true everywhere *except* A. Thus g is false at A, and f is valid. A conventional PROLOG interpreter, however, will not respond sensibly to the intermediate query `g(x)`, and may not return this result.[4]

Finally, we note that if we add the premise `g(A)` to the above program fragment, the query `f` should fail. We shall see shortly that this particular nonmonotonic feature of the problem we are considering places further constraints on procedures designed to respond correctly to queries of this sort.

The outline of this paper is as follows: We begin in the next section by examining briefly some previous efforts to deal with this problem and discussing, in informal terms, our solution to it. In Section 3, we discuss the nature of the problem from a formal point of view, formalizing the idea of an answer "schema" that can be used to report a partial solution to an unbound negative subgoal, and describing in precise terms conditions under which such answer schemas are satisfactory responses to a PROLOG query. In Section 4, we address some technical difficulties arising out of this definition. Our procedural suggestions for solving the problem begin in Section 5, where we discuss the case where the investigation of the negative subgoal involves no additional default assumptions. In Section 6, we go on to extend our methods in a way that allows us to drop this assumption; in Section 7, we

[3]Indeed, recent work of Przymusinska and Przymusinski describing the semantics of logic programs has suggested dropping the domain closure assumption and not restricting our attention to Herbrand models of the theory in question [19].

[4]In fact, the correct result may be returned, but for completely the wrong reason – since `h(x)` succeeds for `x` bound to `A`, `not(g(x))` fails (!) and `f` therefore succeeds.

consider problems arising from interactions between positive and negative subgoals with free variables.

Section 8 shows our techniques at work by discussing a simple planning example in some depth; this example is one that cannot be treated by existing descriptions of negative subgoals containing unbound variables. Concluding remarks are contained in Section 9.

2 Previous work

PROLOG's assumption that negative subgoals be ground has been addressed in the literature only very recently. The various suggestions that have been made [3, 4, 7, 9, 17, 20] all involve finding precise solutions to subgoals of this type, generally by enumerating or characterizing in some other way the exceptions to a subgoal such as $\neg g(x)$ arising out of (1). Chan suggests in [3] that the exceptions to the subgoal $\neg g(x)$ be characterized by running the interpreter to exhaustion on the positive subgoal $g(x)$ and collecting the results, while Foo et al. in [9] make the extraordinary suggestion that the free variable (x, in this case) be bound to all of its reasonable instantiations and the theorem prover be invoked on each of the resulting subgoals $\neg g(A_1), \ldots, \neg g(A_n)$. Chan goes on to describe in some detail how information of the form, "$\neg g(x)$ holds for all x except $x \in \{C_1, \ldots, C_k\}$" can be manipulated when solving subsequent subgoals. This work is quite reminiscent of earlier analysis due to Colmerauer and Kunen [7, 17].

Even for finite domains, Foo's suggestion seems completely unworkable for problems of interesting size, while Chan's approach, known as *constructive negation*, is likely to run into trouble if there are a large or infinite number of solutions to the positive subgoal $g(x)$, or if these solutions are overly expensive to compute. In [4], Chan attempts to address this difficulty by proposing that the solutions to $g(x)$ not be computed explicitly, but only be characterized in some sharp way that is sufficient to enable subsequent subgoals to be processed.

All of these ideas seem very much at odds with the natural interpretation of PROLOG's negation operator as a nonmonotonic one [12]. The work on nonmonotonic reasoning has made it clear that on many problems involving negation as failure, it is impractical or impossible to either enumerate the exceptions to a default rule or to characterize them in any precise way. As an example, when trying to decide whether or not some particular bird can fly, it is impractical to either compare it to all the birds known not to fly (the chicken we had for dinner, the penguin I saw in the zoo yesterday, and so on) or to compare it to all of the *types* of birds known not to fly (ostriches, emus, birds with their feet set in concrete, and so on). The first approach corresponds to constructive negation as Chan originally described it in [3], while the second is similar to his more recent suggestions.

Here is another example: Suppose that I am leaving for work in the morning, and notice that my wife has left her car parked behind mine in the driveway. To get to work, I form the plan of moving her car, and then getting into mine and driving to work. But if my intention is to move her car to some particular location l, I encounter the negative subgoal

$$\neg\texttt{bad-choice}(l),$$

where $\texttt{bad-choice}(l)$ indicates that moving her car to l won't help me achieve my goal of getting to work.

In practice, there are many poor choices for l. Moving the car an inch or so is unlikely to help me get out of the driveway. Driving it into the neighbor's living room is also probably unwise, since the ensuing ruckus is likely to delay me considerably. I don't bother enumerating all of these possible difficulties, or even considering them. I simply plan to get in my wife's car, move it, and then drive to work. Provided that I don't pick some pathological choice for her car's new location, everything will be fine.

This idea is the essence of the approach that we will propose in this paper. Roughly speaking, given a negative subgoal containing a free variable, we will replace the variable with a Skolem constant and then evaluate the resulting (ground) subgoal. If this Skolemized subgoal succeeds, we will interpret this as meaning that the original negative subgoal $\neg g(x)$ succeeds for all "nonpathological" choices of x. In order to make these ideas more precise, we will do the following:

1. First, we will formalize the idea of a solution *schema* that holds for "almost" all of its ground instantiations.

2. The bulk of the paper will present a procedure that produces such solution schemas for queries involving negative subgoals with free variables.

3. Finally, we will present a detailed example of a simple planning problem that has features of the sort we are discussing. We will argue that the approach we are proposing is both more natural and more efficient than either the conventional one or that of constructive negation.

3 Solution schemas

We begin by formalizing our suggestion that when presented with a nonground default query for which it may not be reasonable to return either a list of all possible answers or a list of all exceptions to a possible answer, it may well be reasonable to return an answer that holds "in general."[5]

To make precise the notion of a set of exceptions that is "small" compared to the set of instances of the schema for which the goal in fact holds, we begin by augmenting our language

[5]Note that it does *not* suffice to say that an answer schema holds in general whenever it is satisfactory except for finitely many bindings of the variables it contains. The reason for this is that in an infinite domain, an expression may be a satisfactory answer schema even if there are infinitely many expressions for which it fails – provided that the set of exceptions is somehow "small" relative to the size of the domain in its entirety. This argument also applies to any domain that includes function symbols.

(which we assume to be finite) to include an infinite number of new constant symbols, which we will denote by s_i as i ranges over the positive integers. We will generally refer to the s_i as *Skolem* constants, since we are requiring that they not appear in our declarative database.

We will make the unique names assumption with regard to these symbols, so that for any two functional expressions $f(x_1, \ldots, x_m)$ and $g(y_1, \ldots, y_n)$ such that some s_i appears in either, we will only have

$$f(x_1, \ldots, x_m) = g(y_1, \ldots, y_n)$$

if $f = g$, $m = n$ and $x_i = y_i$ for each i.[6]

Definition 3.1 *Let f be a functional expression, and σ a binding list. We will say that σ is* nontrivial *for f if the set of ground instances of $f|_\sigma$ is a proper subset of the set of ground instances of f.*

Essentially, σ is nontrivial for f if it gives a nontrivial binding for some variable or variables that appears in f.

Definition 3.2 *Let f be a functional expression, and S a set of ground instances of f. We will say that S is* of measure 0 in f *provided that one of the following conditions holds:*

1. *There is a nontrivial binding σ for f such that every element of S is an instance of $f|_\sigma$, or*
2. *S is the finite union of sets of measure 0 in f.*

If S is a set of ground instances of f, we will say that S is of measure 1 in f *whenever the complement of S in the set of instances of f is of measure 0 in f.*

Our terminology is borrowed from mathematical analysis. Roughly speaking, if f is a functional expression involving n variables, we can think of the ground instances of f as making up an n-dimensional space; after applying a nontrivial binding to f, the ground instances make up at most an $n-1$-dimensional space. Assuming that the set of possible instantiations for a single variable is infinite, any finite union of $n-1$ or smaller dimensional subsets will be small relative to the n-dimensional space that contains them.

As an example, consider the functional expression $g(x)$. The set $S = \{g(A)\}$ consisting of a single ground instance is of measure 0 in the set of all ground instances of $g(x)$ since every element of S is an instance of $g(x)|_{x=A}$.

The reason that we need to make the unique names assumption about expressions involving the Skolem constants is so that we can prove the following:

Proposition 3.3 *Let f be a functional expression. Then there is no set S that is both of measure 0 and of measure 1 in f.*

[6] Note that our assumption that there exist an infinite number of distinct object constants (the Skolem constants s_i) is actually in *conflict* with the domain closure assumption made by Gelfond, Lifschitz and others.

It is this proposition that will give teeth to all of our subsequent results; it would be of little interest to show that the set of solutions to a particular query was of measure 1 in some functional expression $f(x)$ if *all* sets of ground instances of $f(x)$ were in measure 1 in f!

Proof of Proposition 3.3 We prove the result only for f of the specific form $f(x)$; the general case is similar.

Let s_i be any Skolem constant in our language. Note that if σ is a nontrivial binding for $f(x)$ such that $f(s_i)$ is an instance of $f(x)|_\sigma$, then σ must be the binding list that binds x to s_i. It now follows from the unique-names assumption that for *any* nontrivial binding list σ, the set of ground instances of $f(x)|_\sigma$ can contain at most a single element of the form $f(s_i)$ for some s_i.

As a result, any set of measure 0 in f can contain only finitely many elements of the form $f(s_i)$. Since there are infinitely many such elements, there is no set S that is of measure 0 in f and whose complement is of measure 0 in f. □

Proposition 3.4 *Let $q(x)$ be a query, f a functional expression and α a ground instance of f in which all of the variables have been bound to unique Skolem constants. Then the set of instances of f for which q succeeds is of measure 1 in f if $q(\alpha)$ succeeds and of measure 0 in f otherwise.*

Proof. Suppose we denote by S the set of all instances of f in which the variables have been bound to unique Skolem constants. The proposition now follows from the following two observations:

1. $q(\alpha)$ succeeds if and only if q succeeds for every element of S, and
2. S is of measure 1 in f.

The first of these is immediate, since the Skolem constants do not appears elsewhere in the database and the truth or falsity of any particular element of S must therefore be independent of precisely which Skolem constants it contains.

For the second, we suppose that f is of the form $f(x, y)$; the general case is no harder. Now for any ground instance of f not to be in S, it must be of one of the following three forms:

1. $f(s_i, s_i)$ (i.e., x and y are not replaced with *distinct* Skolem constants),
2. An expression where either x or y has been bound to an object constant that is not one of the Skolem constants s_i, or
3. An expression where either x or y has been bound to a functional expression.

Note, however, that each of the above three sets of ground instances is of measure 0 in f. The first is given by the nontrivial binding that binds x to y. The second is the finite union of the bindings that bind either x or y to each of the non-Skolem constant symbols in our language, and the third is a similar union of the result of binding x or y to an element of the particular functional form $f_i(a_1, \ldots, a_k)$ where f_i is one of the function constants in the language (of which there are only finitely many). It follows that the complement of S is the union of three sets of measure 0 in f, and is therefore of measure 0 in f itself. Thus S is of measure 1 in f. □

This proposition provides the fundamental justification for the ideas in this paper. Essentially, our approach will be to return answers involving Skolem constants when responding to PROLOG queries. Having done so, we can then reconstruct the functional expression f appearing in the proposition by replacing the Skolem constants in the answer with distinct variables, and then will know that except for a set of measure 0, all ground instances of this functional expression will satisfy the original query. In practice, we may wish to return the answer with the Skolem constants included, since variables appearing in this expression can be bound to *any* constant in the language without affecting the validity of the result.

As an example, consider the logic program

```
f(x,y) :- not(g(y)).
g(A).
```

Our response to the query $f(x, y)$ will be to bind y to the Skolem constant s_1, obtaining $f(x, s_1)$. Since x is unbound in this expression, it indicates that for any choice of x whatsoever, the query succeeds for all choices of y outside a set of measure 0 (in this case, all y except $y = A$).

In order to make some of these ideas precise, we make the following definitions:

Definition 3.5 *Let f be a ground functional expression. We will denote by $\overline{f}$ the result of replacing distinct Skolem constants in f with distinct variables. Having done so, any ground instantiation of $\overline{f}$ will be called a* Skolem instance of f.

As an example, suppose that f is the above expression $f(x, s_1)$. Then $\overline{f}$ is the expression $f(x, y)$ and any ground instance of this expression (such as $f(A, B)$ or $f(s_1, B)$) is a Skolem instance of $f(x, s_1)$.

Definition 3.6 *Given a query $q(x)$ and a ground functional expression f, we will say that f is an* acceptable response to the query *if the set of solutions to the query is of measure 1 in $\overline{f}$. Given a query $q(x)$ and an arbitrary functional expression f, we will say that f is an* acceptable response to the query *if every ground instantiation of f is an acceptable response to the query.*

An acceptable response f to the query will be called Skolem optimal *if f is ground and there is no other acceptable ground response of which f is a nontrivial Skolem instance. The response will be called* optimal *if there is no other acceptable response (ground or otherwise) of which f is a nontrivial instance (Skolem or otherwise).*

In the above example, $f(x, s_1)$ is the unique optimal response. $f(s_2, s_1)$ is Skolem optimal but not optimal; $f(s_2, g(s_1))$ is not Skolem optimal because $g(s_1)$ is a nontrivial Skolem instance of s_1. Expressions such as $f(x, y)$ are not acceptable responses because the query fails for $y = A$.

Lemma 3.7 *A conventional* PROLOG *interpreter will return optimal responses to queries that do not generate negative subgoals with free variables.*

Proof. In the absence of Skolem constants, a response that is acceptable according to Definition 3.6 is acceptable by the usual standard. An optimal response is simply any most general response to the query. ◻

With this formal machinery behind us, we are now in a position to state our intention in this paper: *Given a procedure that produces optimal responses to* PROLOG *queries that do not generate negative subgoals with free variables, we will describe a procedure that produces optimal responses to all such queries, whether they generate such subgoals or not.*

4 Optimal responses from Skolem optimal ones

We will achieve the above goal in two stages. The bulk of the paper will be devoted to describing a procedure that produces Skolem optimal responses to arbitrary queries; in this section, we show how to turn Skolem optimal responses into optimal ones.

The basic difference between a Skolem optimal response and an optimal one is that while a Skolem optimal response indicates that "most" of its Skolem instances are solutions to the original query, a response in which the Skolem constants have been replaced with variables indicates that *every* instance is a solution to the original query. This suggests that in order to change a Skolem optimal solution to an optimal one, we need simply check to see if the various Skolem constants can be replaced with variables:

Procedure 4.1 *Suppose that we have some method for generating Skolem optimal responses to* PROLOG *queries, and that the response α has been generated in response to the query q. Now renumber the Skolem constants appearing in α so that they are $s_1, \ldots, s_n$ and do the following:*

1. *Set $i = 1$.*
2. *Let β be the result of replacing all appearances of s_i in α with a variable that does not appear elsewhere in α.*
3. *Invoke the interpreter on the query $\neg q(\beta)$.*
4. *If this query fails, set $\alpha = \beta$. If it succeeds, leave α unchanged.*
5. *If $i < n$, increment i and return to step 2. Otherwise, return α as a solution to the original query.*

To see this procedure working, we return to the example of the previous section, where we considered the PROLOG fragment

```
f(x,y) :- not(g(y)).
g(A).
```

Suppose that in response to the query $f(x, y)$, we are given the Skolem optimal answer $f(s_1, s_2)$.

We begin by replacing s_1 with x, generating the candidate solution $f(x, s_2)$. The interpreter on $\neg f(x, s_2)$ fails, since the positive version of this query, $f(x, s_2)$, succeeds for every choice of x. This means that our solution remains acceptable if we do not replace x with the original Skolem constant s_1.

We now turn to s_2, replacing it with y to get the new candidate solution $f(x, y)$. Now, however, the query $\neg f(x, y)$ succeeds for some ground x and y; as an example, $\neg f(B, A)$ is a consequence of the program being considered. We therefore replace s_2 in the original expression, returning $f(x, s_2)$ as our response to the original query.

In some cases, the answer returned by Procedure 4.1 may depend upon the order in which the Skolem constants are considered. As an example, consider the query $\neg f(x, y)$ in a situation where the database contains the single fact $f(A, B)$. A Skolem optimal solution is $\neg f(s_1, s_2)$; depending on which Skolem variable is replaced first, either of the two optimal solutions $\neg f(x, s_2)$ and $\neg f(s_1, y)$ may be returned.

Proposition 4.2 *Given that our interpreter returns Skolem optimal responses to arbitrary queries, Procedure 4.1 will return optimal responses to arbitrary queries. Furthermore, any specific optimal responses can be produced in this fashion.*

Proof. For the first part, we need to show that the β returned by Procedure 4.1 is acceptable, and that it is optimal. To see that β is acceptable, we need simply note that when a Skolem constant in α is replaced with a variable in the construction of β, the subgoal in step 3 of the procedure serves to ensure that there is no binding of this variable for which the original query fails.

Showing that β is optimal is harder, although only slightly so. If β were *not* optimal, there would be some other acceptable response γ of which β was an instance. Now if we replace every variable in γ with a new Skolem constant to obtain the ground expression γ', it follows that γ' will necessarily be a Skolem optimal solution to the original query. Since the α originally provided as input to Procedure 4.1 is also Skolem optimal, it follows that γ' and α are identical up to renaming of Skolem constants.

It follows from this that γ can differ from β only in that γ has variables in some positions in which β has a Skolem constant. Suppose that s_i is the Skolem constant in β that has been replaced in γ with a variable x. There are now two possibilities:

1. If not every instance of s_i in β has been replaced with x, then the result of binding x to a new Skolem constant will be a Skolem optimal response to the original query of which α is a nontrivial Skolem instance.

2. If every instance of s_i in β has been replaced with x, it follows that when s_i was considered by Procedure 4.1, the subgoal considered in step 3 of the procedure would

have failed (since γ is an acceptable response to the query), and s_i would have been rebound to a variable in the construction of β.

In either case there is a contradiction, and this direction of the proof is complete.

The converse is easier. If β is an optimal response to the original query, then the result β' of replacing the variables in β with Skolem constants will be Skolem optimal, and Procedure 4.1 can be used to reproduce β from β'. □

5 Subgoals with monotonic justifications

Let us now return to the more interesting problem of constructing a procedure that generates Skolem optimal responses to arbitrary queries. Suppose that we have begun with some query q that has been reduced to a collection of subgoals, some of which involve free variables in negative clauses. Let us denote such a subgoal by $\neg s(x)$. As we have seen, there are now four distinct possibilities:

1. $\neg s(x)$ holds for all x.
2. Although $\neg s(x)$ does not hold for all x, it holds for a set of measure 1 in the collection of possible x (presumably because of the PROLOG negation-as-failure rule applied to s).
3. $s(x)$ holds for all x.
4. Although $s(x)$ does not hold for all x, it holds for a set of measure 1 in the collection of all possible x, so that $\neg s(x)$ holds only for a set of measure 0.

The fourth possibility is that in which s is itself a consequence of other nonmonotonic assumptions. For the time being, we will assume that this is not the case.

Proposition 5.1 *Let $\neg s(x)$ be a negative subgoal containing free variables, but such that either the set of instantiations of x for which the subgoal succeeds is of measure 1, or the subgoal fails for every instantiation of x. Now if x' is the result of replacing the variables in x with distinct Skolem constants, then:*

1. *If the (ground) query $\neg s(x')$ fails, then so does $\neg s(x)$.*
2. *If $\neg s(x')$ succeeds, then x' is a Skolem optimal response to the original subgoal.*

Proof. Suppose first that $\neg s(x')$ fails. Then we know from Proposition 3.4 that the set of solutions to $\neg s(x)$ is of measure 0, and therefore (by our assumption) that $\neg s(x)$ fails for all instantiations of x.

Alternatively, if $\neg s(x')$ succeeds, then since replacing the Skolem constants in $\neg s(x')$ with variables reproduces $s(x)$ itself, it is clear that x' is a Skolem optimal solution to the original query. □

As an example, we repeat the fragment of the previous section:

```
f(x,y) :- not(g(y)).
g(A).
```

When presented with the query $f(x, y)$, we generate the negative subgoal $\neg g(y)$. Skolemizing y gives us the ground negative subgoal $\neg g(s_1)$, which succeeds. We therefore bind y to s_1 in the original query to obtain the Skolem optimal response $f(x, s_1)$. The response is in fact optimal in this case; we shall see a somewhat more interesting example in Section 8.

6 Subgoals with nonmonotonic justifications

Imagine now that we were to use the techniques of the previous section to attempt to find a solution to the query f in the following situation:

```
f :- not(g(x)).
g(x) :- not(h(x)).
h(A).
```

The goal f generates the negative subgoal $\neg g(x)$. The instantiation $\neg g(s_1)$ fails, although the instantiation $\neg g(A)$ would succeed.

In terms of the discussion in the previous section, we have a negative subgoal $\neg g(x)$ for which we can find a proof of the Skolemized version $g(s_1)$. It follows that the set of x for which the original subgoal should succeed is of measure 0, but it does *not* follow that there is no solution to the subgoal at all (as we assumed earlier).

What we need to do is to understand the failure of the Skolemized subgoal in a way that enables us to identify the instantiation $x = A$ as a potential exception. Ideally, we would like to do this in a way that avoids repeating the search involved in the proof of $g(s_1)$.

We would also like to proceed in a way that allows us to generate and test possible exceptions to the proof of $g(x)$ singly. As an example, suppose that $h(B)$ also appeared in the above database. It is important that we be able to generate and consider the possible exception $x = A$ *before* generating the possible exception $x = B$. The reason for this is that it may be computationally expensive to find such a potential exception (since doing so involves solving the subgoal $h(x)$), and we need to be able to treat such exceptions individually as backtrack points, rather than generating them all when the subgoal is encountered.[7]

Achieving this will not be difficult if we can identify a new subgoal $t(x)$ such that solutions to this subgoal are candidate solutions to the original negative subgoal $\neg s(x)$. In the example we are considering, for example, we have $\neg s(x) = \neg g(x)$ and would expect to have $t(x) = h(x)$.

To understand this, suppose that we have encountered a negative subgoal $\neg s(x)$, and that the instantiated version of this subgoal, $\neg s(s_1)$, has failed.

[7]Once again, we see the difference between our approach and Chan's constructive negation. By treating the solutions to $g(x)$ singly, we ensure that the addition of the axiom $h(B)$ to our database does not adversely affect the time needed to respond positively to the query f; this property is not shared by constructive negation, which must (at least on the face of it) enumerate all of the solutions to $g(x)$ before proceeding.

In order for $\neg s(s_1)$ to fail, there must be some proof of $s(s_1)$. This proof will appeal to the negation-as-failure rule applied to some predicates

$$p_1(s_1), \ldots, p_n(s_1)$$

(perhaps with $n = 0$). It follows from this that

$$\neg p_1(x) \wedge \cdots \wedge \neg p_n(x) \rightarrow s(x) \tag{2}$$

in the conventional sense, without appealing to any additional nonmonotonic assumptions. It follows from *this* that exceptions to the nonground query $s(x)$ will necessarily be solutions to the new subgoal

$$p_1(x) \vee \cdots \vee p_n(x).$$

Note that if the proof of $s(s_1)$ succeeded without appealing to the negation-as-failure rule at all, so that $\forall x.s(x)$ were entailed by our database, we would have $n = 0$ and the new subgoal, an empty disjunction, would be tautologically false. The upshot of this is that we will not attempt to overturn any proofs that did not use the negation-as-failure rule in the first place.

Slightly more generally, there will be a variety of proofs of $s(s_1)$, where the ith proof depends on negation-as-failure assumptions about $p_{i1}, \ldots, p_{in_i}$. Similarly to (2), we can conclude that

$$\bigvee_i \bigwedge_j \neg p_{ij} \rightarrow s(s_1),$$

so that any solution to the original negative subgoal $\neg s(x)$ must be a solution to the new subgoal

$$\neg \bigvee_i \bigwedge_j \neg p_{ij} = \bigwedge_i \bigvee_j p_{ij}. \tag{3}$$

Finally, we note that solutions to the subgoal appearing in (3) still need to be checked as solutions to the original negative subgoal. This is because there may be facts in the database that although not involved in the failure of the original Skolemized subgoal, nevertheless invalidate the instantiations suggested by the consideration of (3).

Here, then, is the procedure for the evaluation of a negative subgoal with free variables:

Procedure 6.1 *To evaluate a negative subgoal $\neg s(x_1, \ldots, x_k, C_1, \ldots, C_l)$, where the x_i are free variables and the C_j are ground terms:*

1. *Replace the variables with Skolem constants and invoke the interpreter on the result.*

2. *If the Skolemized subgoal succeeds, return it as the solution to the original negative subgoal.*

3. *If the Skolemized subgoal fails, identify the negation-as-failure assumptions responsible for the failure, and construct the new subgoal appearing in (3).*

4. *For each solution σ to this new subgoal, where σ is a binding list for some of the variables appearing in the original negative subgoal, invoke the procedure recursively on the new negative subgoal $\neg s(x)|_\sigma$.*

As an example, we return to the program fragment that began this section. Here it is again:

```
f :- not(g(x)).
g(x) :- not(h(x)).
h(A).
```

The goal f generates the negative subgoal $\neg g(x)$; skolemizing gives the subgoal $\neg g(s_1)$. Since attempting to prove $h(s_1)$ fails, $g(s_1)$ succeeds and the Skolemized subgoal $\neg g(s_1)$ fails. But we note that the proof of $g(s_1)$ involves applying the negation-as-failure rule to $h(s_1)$, and this in turn leads us to consider the positive subgoal $h(x)$. This subgoal succeeds with x bound to A, so we reconsider the original negative subgoal $\neg g(x)$, binding x not to s_1 but to A. Since $h(A)$ succeeds, $\neg h(A)$ fails, and $g(A)$ fails. We therefore return $x = A$ as a solution to the original negative subgoal, and the query f succeeds.

If, on the other hand, we augment the above database with the statement `g(A)`, then the instantiated negative subgoal $\neg g(A)$ fails, and the original query fails as well.

Here is a more subtle example, where the query is `f(x,y)`:

```
f(x,y) :- not(g(x,y)).
g(x,y) :- not(h(x)).
h(A).
g(A,y) :- not(j(y)).
j(B).
```

Now the negative subgoal $\neg g(x, y)$ instantiates to $\neg g(s_1, s_2)$. This subgoal fails, since $\neg h(s_1)$ succeeds. But the success involves the application of the negation-as-failure rule to h, so we are led to the positive subgoal $h(x)$. This succeeds for x bound to A, so we bind x in the original subgoal to obtain $\neg g(A, y)$.

Applying the procedure recursively, we Skolemize $\neg g(A, y)$ to obtain $\neg g(A, s_3)$. This subgoal also fails, since $\neg j(s_3)$ succeeds. Once again, the failure is due to the application of the negation-as-failure rule (to j this time), so we consider the new subgoal $j(y)$, which succeeds for y bound to B. We bind y to B in the subgoal $\neg g(A, y)$ to obtain $\neg g(A, B)$. This query finally succeeds, allowing us to return $x = A, y = B$ as a solution to the original query.

Proposition 6.2 *In a language without function symbols, suppose that our interpreter always terminates on positive subgoals and ground queries. Then provided that every variable appearing in the body of a database rule also appears in its head, Procedure 6.1 will terminate as well.*

Proof. The procedure can fail to terminate only if it continues to call itself recursively. But note that the positive subgoals constructed in step 3 of the procedure can never succeed without instantiating at least one of their variables. (If they did, the original Skolemized subgoal would have succeeded as well.)

It follows that the new subgoal passed recursively to the procedure will have fewer free variables than its predecessor, so that the procedure must terminate eventually. □

This result fails if the assumptions are weakened. The reason is that the positive subgoal introduced in step 3 of the procedure may itself spawn negative subgoals, and the optimal responses to these subgoals may introduce new variables into the original subgoal and thereby cause the evaluation procedure to fail to terminate. As we will see in Section 8, simple planning problems tend to contain features of just this sort.

More important than Proposition 6.2, of course, is the following:

Proposition 6.3 *Let $\neg s(x)$ be a negative subgoal containing free variables. Then the result of applying Procedure 6.1 to $\neg s(x)$ is a Skolem optimal response to it, and any such response can be generated in this fashion.*

Proof. The proof is in fact quite straightforward. We know from Proposition 5.1 that as soon as the Skolemized subgoal has succeeded, a Skolem optimal response to the bound subgoal $\neg s(x)|_\sigma$ will have been produced. Since we know that *every* solution to the original negative subgoal solves $\neg s(x)|_\sigma$ for one of the σ's produced in the final step of the procedure, it follows from the Skolem optimality of these σ's that any Skolem optimal solution to the modified subgoal will be a Skolem optimal solution to the original one. □

7 Multiple subgoals

There is one additional subtlety that appears if Procedure 6.1 is to be used in practice. Consider the following PROLOG fragment:

```
f(x) :- g(x),not(h(x)).
g(A).
h(B).
```

When presented with the query $f(x)$, the subgoal $g(x)$ is evaluated, binding x to A. This leads to the ground negative subgoal $\neg h(A)$, which succeeds, so that the original goal returns $f(A)$. But suppose that the order of the two clauses in the definition of f had been reversed:

```
f(x) :- not(h(x)),g(x).
g(A).
h(B).
```

Now the negative subgoal $\neg h(x)$ succeeds by binding x to the Skolem constant s_1, and the positive subgoal $g(s_1)$ fails. The problem is that the positive subgoal should be looking for a solution that is a *Skolem* instance of $g(s_1)$, as opposed to an instantiation of it in the conventional sense.

There are two ways around this difficulty. The simpler one is merely to always consider positive subgoals before negative ones. In some instances, however, this may be overly restrictive:

```
f(x,y) :- not(g(x,y)), hard(x).
g(x,y) :- not(h(x)).
h(A).
g(B,y).
hard(B).
hard(A).
```

We are supposing that the subgoal **hard(x)** succeeds by binding x to A or B, but that the evaluation of this subgoal is computationally expensive. Since applying Procedure 6.1 to the subgoal $\neg g(x, y)$ also binds x to A, it is important that the subgoals be evaluated in the order in which they are supplied.

The alternative approach overcomes this difficulty by replacing the Skolem constants appearing in an arbitrary positive subgoal s with variables to obtain $\overline{s}$ as in Definition 3.6. The resulting subgoal is then solved and the Skolem constants replaced. Any solutions to $\overline{s}$ that bind newly introduced variables away from Skolem constants will need to be checked to ensure that they remain solutions to the negative subgoals that introduced the Skolem constants in the first place. It is clear that this procedure is correct, since all it is really doing is ensuring that the matcher recognizes Skolem instances of the solution to a particular negative subgoal.

8 An extended example: Planning

In order to show our ideas at work, we will now consider in some detail slightly more difficult examples of the problems we have been discussing.

The application we will consider is that of planning and reasoning about action, working with a truly *nonmonotonic* description of our domain. We do this in part because it is the nonmonotonic nature of the description that leads to negative subgoals with free variables, but also because it appears [14, 15] that a nonmonotonic description will be the only computationally tractable one as the domains encountered by general-purpose planning systems grow increasingly complex.

Rather than discuss complex domains, however, we will consider only simple blocks worlds problems. The positions of objects will be described using a predicate $\mathtt{on}(b, l)$, indicating that the block b is at the location l.

We will work with a simple version of the situation calculus, denoting the fact that some sentence p holds in a situation s by writing

$$\mathtt{holds}(p, s).$$

A situation will be written as an ordered sequence of actions

$$s = \langle a_n, \ldots, a_1 \rangle.$$

(We write the last action first so that we can append new actions to the front of the sequence.) The initial situation is written simply as $\langle\rangle$, corresponding to the empty sequence of actions.

We will denote by $\langle a|s \rangle$ the result of performing the action a in the situation s. The frame axiom, a default rule saying that things tend to stay the same as time goes by, is now written as:

$$\mathtt{holds}(p, s) \wedge \neg\mathtt{ab}(a, p, s) \rightarrow \mathtt{holds}(p, \langle a|s \rangle). \tag{4}$$

In other words, if p holds in a situation s, and the action a is not abnormal in that its execution potentially reverses p, then p will continue to hold in the resulting state $\langle a|s \rangle$.

There is a single action in this domain, $\mathtt{move}(b, l)$, which relocates a block b at a location l. The preconditions for this action are that the block being moved and the target location both be clear. Formally, we have:

$$\mathtt{holds}(\mathtt{clear}(b), s) \wedge \mathtt{holds}(\mathtt{clear}(l), s) \rightarrow \mathtt{holds}(\mathtt{on}(b, l), \langle \mathtt{move}(b, l)|s \rangle) \tag{5}$$

together with the axiom defining `clear` as:

$$\mathtt{holds}(\mathtt{clear}(x), s) \equiv \nexists y . \mathtt{holds}(\mathtt{on}(y, x), s). \tag{6}$$

Of course, the existential quantifier appearing in (6) forces us to modify this description somewhat if we are to use a PROLOG-like theorem prover. Here is a suitable redescription[8]:

```
holds(p,s) :- append(s1,s2,s), holds(p,s2), not(ab(s1,p,s2)).
holds(on(b,l),[move(b,l)|s]) :- not(occupied(b,s)),
                                not(occupied(l,s)).
occupied(x,s) :- holds(on(y,x),s).
```

Note that we have modified the frame axiom somewhat so that we can conclude at a single stroke that the sentence p will be preserved through the action sequence s_1. In addition to being more efficient than the previous description, this modification is necessary if negative subgoals involving the frame axiom are to be dealt with properly. Taking s_2 to be the initial situation $\langle\rangle$, we get as a consequence of this modified axiom that

$$\mathtt{holds}(p, \langle\rangle) \wedge \neg\mathtt{ab}(s, p, \langle\rangle) \rightarrow \mathtt{holds}(p, s). \tag{7}$$

In other words, in the absence of an abnormality, things true in the initial situation will be true in all subsequent situations.

Note also that we cannot rewrite the second PROLOG rule as

[8]As we shall see shortly, we will also need the contrapositive version of the frame axiom (4), since we will need to be able to derive (for example) $\mathtt{ab}(s_2, p, s_1)$ if p does not persist through the execution of the action sequence s_2 in the situation s_1.

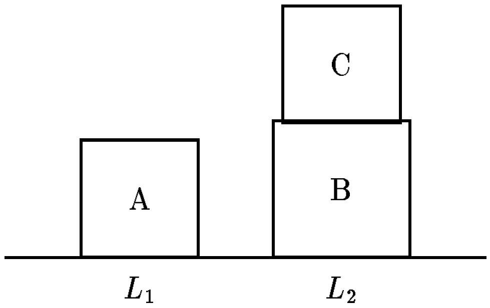

Figure 1: Get either B or C onto A

```
holds(on(b,l),[move(b,l)|s]) :- not(holds(on(x,b),s),
                                not(holds(on(y,l),s).
```

since the semantics we have assigned to this expression are different from those of the original one.[9] (The above rule allows us to move b to l if there is *any* block x that is not on b, and *any* block y that is not already at l.)

The initial situation is that depicted in Figure 1, and is described by the following axioms:

```
holds(on(A,L1),[])
holds(on(B,L2),[])
holds(on(C,B),[])
```

The blocks A and B are both on the table, and C is on top of B. The goals we will consider are those of getting either C or B on top of A.

Consider first the goal of getting C on A, corresponding to the PROLOG query

```
holds(on(C,A),s).
```

Resolving this goal against the rule describing the `move` action, we obtain the pair of subgoals

```
not(occupied(C,t))
```

and

```
not(occupied(A,t)).                (8)
```

Instantiating t to a Skolem constant in the first subgoal, we obtain

```
not(occupied(C,S1)),
```

and this subgoal succeeds. In addition, applying Procedure 4.1, we see that this is an optimal solution to the original negative subgoal.

[9]The semantics are in fact the same under the conventional interpretation described in footnote 2. Although this is an argument for the standard interpretation, there is no way to express an axiom such as (1) using standard methods.

In accord with the ideas of Section 7, we replace the Skolem constant s_1 with the variable t when processing the second subgoal (8). This subgoal also succeeds with t bound to s_1, so that we finally conclude that the original query `holds(on(C,A),s)` holds for s bound to `[move(C,A)|S1]`.

We now view the proof process as complete, returning the answer schema

$$\texttt{[move(C,A)|S1]}, \tag{9}$$

where the Skolem constant s_1 denotes an unspecified situation (i.e., sequence of actions). If we wish, we can substitute the empty action sequence `[]` for the Skolem constant s_1, producing the simpler result binding s to `[move(C,A)]`. Of course, this new answer will need to be tested to make sure that the default rules leading to the Skolemized conclusion are not violated when we replace the Skolem constant with a particularly convenient value.

If our theorem prover can deal with conjunctive subgoals, we can streamline this procedure somewhat be considering the negative subgoals

$$\texttt{not(occupied(C,t))}$$

and

$$\texttt{not(occupied(A,t))}$$

simultaneously. Since the conjoined negative subgoal

$$\neg\texttt{occupied}(C,t) \wedge \neg\texttt{occupied}(A,t)$$

succeeds by binding t to s_1, we avoid one invocation of Procedure 4.1.

Note that constructive negation is completely incapable of dealing with this problem. If it were our intention to return a "complete" answer to the original query, we would have to modify (9) in some way so that it included information about those situations s' that violated the frame axiom (7) and could therefore not be instantiated into the solution schema given by (9).

Unfortunately, this is completely unsuitable for problems such as the one we are considering – how are we possibly to enumerate *all* of the states s' in which something is on top of C or of A? There are infinitely many such states, arrived at by infinitely many distinct action sequences. In this particular case, not only is the schema (9) a reasonable response to the original query (indeed, it is optimal in the sense of Definition 3.6); it is the only response available. Applying the techniques of previous authors will fail to return an answer at all.

Let us now turn to the more difficult of the two planning problems, corresponding to the goal `holds(on(B,A),s)`. Resolving it with the description of the `move` action leads to the compound negative subgoal

$$\neg\texttt{occupied}(B,t) \wedge \neg\texttt{occupied}(A,t).$$

Instantiating t produces

$$\neg\texttt{occupied}(B,s_1) \wedge \neg\texttt{occupied}(A,s_1).$$

Although the second clause succeeds, the first clause here fails, since `occupied(B,S1)` is a consequence of the default assumption

```
not(ab(S1,on(C,B),[])).
```

Procedure 6.1 now leads to the construction of the new positive subgoal `ab(t,on(C,B),[])`.

In order to solve this subgoal, we need some way to conclude that an action sequence t is abnormal. This involves a contrapositive version of the frame axiom:

```
ab(s1,p,s2) :- holds(p,s2), append(s1,s2,s), not(holds(p,s)).     (10)
```

Note, however, that the negation appearing in this axiom should be interpreted not using the negation-as-failure rule, but in terms of an ability to actually prove `not(holds(p,s))` directly. After all, we are only prepared to conclude that an action sequence is abnormal if we can prove *without* using negation-as-failure that some sentence has changed from true to false when the action sequence is executed. This leads us to rewrite (10) as:

```
ab(s1,p,s2) :- holds(p,s2), append(s1,s2,s), no-holds(p,s))
```

where `no-holds` is a predicate introduced for this purpose. Given this axiom, the subgoal `ab(t,on(C,B),[])` now leads to the new subgoal `no-holds(on(C,B),t))`. Informally, we can show that an action sequence is abnormal with respect to C being on B provided that we can demonstrate that it actively causes `on(C,B)` to become false.

At this point, we need an axiom stating conditions under which C will no longer be on B – nowhere in our domain description, for example, is there an axiom saying that a block can be in only one place at a time! The following is suitable:

```
no-holds(on(x,y),s) :- holds(on(x,z),s), not(y=z).
```

This axiom does indeed say that a block can be in only one place at a time. Resolving it with our subgoal `no-holds(on(C,B),t)` leads to `t=[move(C,z)|u]`, together with the compound negative subgoal

$$\neg\texttt{occupied}(C,u) \wedge \neg\texttt{occupied}(z,u) \wedge (B \neq z).$$

Instantiating u to s_2 and z to s_3, this subgoal succeeds.

Inserting `t=[move(B,A),move(C,S3)|S2]` into our original subgoal, we get

$$\neg\texttt{occupied}(B,\langle\texttt{move}(C,s_3)|s_2\rangle) \wedge \neg\texttt{occupied}(A,\langle\texttt{move}(C,s_3)|s_2\rangle).$$

This subgoal succeeds, and we therefore return as a final answer the action sequence

```
[move(B,A),move(C,S3)|S2].     (11)
```

It is only an answer schema that is returned, of course. The interpretation of (11) is as follows: To get B onto A, perform an arbitrary (but hopefully nonpathological) sequence of actions s_2. Then move C somewhere, and put B on A. The choices A and B will be pathological instantiations of s_3, as it turns out.

8.1 Implementation concerns and results

Examining step 3 of Procedure 6.1, we see that when the proof of the negated Skolemized subgoal succeeds (so that the proof of the Skolemized subgoal itself fails), we need to know which negation-as-failure assumptions were involved in the successful proof.

There are a variety of possible ways to achieve this when implementing the ideas we have presented. The simplest is probably to use an ATMS [8] when evaluating the negated subgoal. As discussed in [13, 21], an ATMS can be used to return the minimal set of assumptions needed to support a given conclusion; such a set is just what is needed to construct the new subgoal appearing in (3). (Reiter and de Kleer refer to such a minimal set as a *prime implicant.*)

This observation was used to incorporate the work we have described into the multivalued theorem prover discussed in [13]; the reason this theorem prover was used is that it already combines techniques from both truth maintenance and default reasoning. The two planning problems we have discussed were then solved using the techniques we have developed. The principal purpose of the implementation was to contrast the time needed by the approach we have presented with the time needed to solve the problems using conventional monotonic techniques such as those discussed in [11]. In both cases, general-purpose planning techniques were used, as opposed to special-purpose methods developed to deal with the blocks-world domain only.

Compromises had to be made in both implementations. In the nonmonotonic case, some of the well-known problems involved in using nonmonotonic techniques to reason about action were encountered [16]. These difficulties were overcome by assigning specific priorities to various default assumptions. The resulting system was indeed able to solve the planning problems we have discussed, and the inference paths were as described earlier. The axiomatization was quite brittle, however, and is unlikely to be useful in dealing with more complex problems in this domain. The recent work in [2] will be able to address these difficulties, but has not yet been combined with the techniques that are the subject of this paper.

In the monotonic case, two compromises were made. The monotonic version of the frame axiom (4) will be something like:

```
holds(on(b,l),[move(b',l')|s]) :- holds(on(b,l),s), not(b=b')
```

which says that moving a block b' will never change the location of a different block b. Unfortunately, this axiom leads to recursive inference paths. It was therefore replaced with two separate versions which could be used for action sequences of length 1 or 2:

```
holds(on(b,l),[move(b',l')]) :- holds(on(b,l),[]), not(b=b').
holds(on(b,l),[move(b',l'),a]) :- holds(on(b,l),[a]), not(b=b').
```

This replacement is sufficient to allow the solution of our two simple problems, but is obviously not strong enough to solve anything more difficult.

The other "compromise" made was that the description used did not include any information regarding the "delete lists" of the actions being considered. If the system needed to know that moving C onto A resulted in C's no longer being on B, it had to prove that by using the fact that C would be on A after the action was performed, and that blocks could only be in one place at a time. This description, although not as efficient as one that explicitly lists all of the positive and negative consequences of an action, seemed reasonable for two reasons. First, no such explicit description was made available to the nonmonotonic planner that was solving the same problems. Second, it appears [14] that it will not in general be *possible* to precompute all of the ramifications of an action in a complex domain.

Finally, the monotonic planner needed to be told of the existence of a new location L_3. Otherwise, it had no place to put C when it wanted to move B onto A!

Both problems were presented to the systems, running on a Symbolics 3620. The speed of the interpreter used was such that both systems were capable of performing approximately 50 inferences/second. The nonmonotonic approach solved the simpler of the two problems in about 4 seconds, and the more complex one in about 6. The monotonic approach solved the simple problem immediately, but took 18 seconds on the more difficult one.

Why should the nonmonotonic system be faster? The reason is simply that the inferences performed by this approach are very much like those taken by a human planner: First, a simple plan is constructed that achieves the goal based on default assumptions about the domain. If some of these default assumptions are found to be invalid, the plan is then modified in such a way that the difficulty is overcome.

In our particular case, the defaults used involved assuming that various blocks were clear. When the assumption that B was clear was found to be invalid in the second planning problem, a new sequence of actions was developed that would cause B to be clear after all, and this patch was spliced into the flawed plan to create an acceptable one.

The monotonic planner was not able to focus its search in this fashion; it had no way of knowing that the naïve plan of simply moving B onto A was in any way "close" to a solution to the problem. It therefore needed to perform a laborious backtrack through its domain description, regressing all of the preconditions to the various actions through all of the actions it was considering, and so on. The result was that it spent a great deal of time working on problems that turned out to have no relevance to the eventual solution it discovered.

There have been many suggestions recently that general-purpose planning must work in the way our nonmonotonic approach proceeded: Potential plans are constructed (or retrieved from a library of such plans), criticized, and modified if necessary to overcome any obstacles that are discovered. There has been little formal work on what it means to criticize a plan, however, and no work at all on the problem of patching a flawed plan. What we have seen here is that both the criticism and recovery processes can be described naturally in terms of the evaluation of negative subgoals with free variables, and that the ideas we have developed can therefore be used to give us a formal handle on a very attractive approach to planning generally.

9 Conclusion

We conclude with some informal remarks regarding the computational nature of the procedure we have described.

Because of the undecidable nature of consistency checking, there are serious computational problems involved in the use of any nonmonotonic description of a domain. Nevertheless, it has been argued elsewhere ([14, 15] is typical) that commonsense reasoning must ultimately depend upon nonmonotonic techniques if we are to model domains complex enough to be interesting.

This paper has discussed one of the issues that will need to be addressed if we are to use nonmonotonic reasoning to construct such a commonsense reasoning system – the appearance of default queries containing free variables. We have presented a procedure for dealing with such queries, shown several fairly simple examples of the procedure in action, and formalized the nature of the answers returned by this procedure. We have also seen that existing proposals for dealing with these queries appear to be unable to address the problems likely to arise in planning and commonsense reasoning.

It might appear that the complexity of the procedure when applied to even the very simplest of planning problems will result in our methods being unsuitable as these problems become more difficult, but this is not the case. The reason is that the work actually done in developing the simple plans we have considered is remarkably parallel to the effort we would expect to make from a commonsense point of view. As a result, even very simple planning problems can be solved *more* efficiently using our techniques than by an appeal to purely monotonic methods.

Acknowledgement

This work has been supported by General Dynamics and by NSF under grant number IRI89-12188. I would like to thank Keith Clark, Vladimir Lifschitz, Teodor Przymusinski and David Smith for many useful discussions, and the members of Formfeed for the patience with which they considered an early version of these ideas.

References

[1] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Proceedings of the 1986 Workshop on Foundations of Deductive Databases and Logic Programming*, 1986.

[2] A. B. Baker and M. L. Ginsberg. Temporal projection and explanation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 906–911, 1989.

[3] D. Chan. Constructive negation based on the completed database. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 111–125, 1988.

[4] D. Chan. An extension of constructive negation and its application in coroutining. In *Logic Programming: Proceedings of the Sixth International Conference and Symposium*, 1989.

[5] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum, New York, 1978.

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

[7] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[8] J. de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:127–162, 1986.

[9] N. Foo, A. Rao, A. Taylor, and A. Walker. Deduced relevant types and constructive negation. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 126–139, 1988.

[10] M. Gelfond and V. Lifschitz. Compiling circumscriptive theories into logic programs. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1988.

[11] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[12] M. L. Ginsberg. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, Los Altos, CA, 1987.

[13] M. L. Ginsberg. Multivalued logics: A uniform approach to reasoning in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[14] M. L. Ginsberg and D. E. Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.

[15] M. L. Ginsberg and D. E. Smith. Reasoning about action II: The qualification problem. *Artificial Intelligence*, 35:311–342, 1988.

[16] S. Hanks and D. McDermott. Nonmonotonic logics and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.

[17] K. Kunen. Answer sets and negation as failure. In *Logic Programming: Proceedings of the Fourth International Conference and Symposium*, pages 219–228, 1987.

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

[19] H. Przymusinska and T. C. Przymusinski. Semantic issues in deductive databases and logic programs. To appear.

[20] T. C. Przymusinski. On constructive negation in logic programming. In preparation.

[21] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 183–188, 1987.