An Abductive Framework for Generalized Logic Programs: Preliminary Report

Gerhard Brewka*

TR-92-049

July 1992

Abstract

We present an abductive semantics for generalized propositional logic programs which defines the meaning of a logic program in terms of its extensions. This approach extends the stable model semantics for normal logic programs in a natural way. The new semantics is equivalent to stable semantics for a logic program P whenever P is normal and has a stable model. The existence of extensions is guaranteed for all normal programs. The semantics can be applied without further modification to generalized logic programs where disjunctions and negation signs may appear in the head of rules. Our approach is based on an idea recently proposed by Konolige for causal reasoning. Instead of maximizing in abduction the set of used hypotheses alone we maximize the union of the used and refuted hypotheses.

*On leave from GMD, Postfach 12 40, 5205 Sankt Augustin, Germany

1 Background and Motivation

In this paper we investigate the relationship between abduction and logic programming.[1] This investigation is interesting for several reasons. Firstly, abduction as a form of nonmonotonic reasoning has gained a lot of interest in recent years, and exploring the relationship between different forms of nonmonotonic reasoning is of interest in itself. Secondly, as we will show in this paper, it is possible to define a simple and elegant extension of Gelfond and Lifschitz's stable model semantics [GL88] based on abduction. This new abductive semantics has the following properties:

- the semantics is equivalent to stable model semantics for programs which possess at least one stable model,
- a program P has a defined meaning unless P considered as a set of inference rules is inconsistent. In particular, normal logic programs without stable model are not meaningless,
- the semantics is, without further modification, applicable to generalized logic programs, that is, logic programs where disjunctions and negation signs may appear in the head of a rule.

We consider all of these properties as highly desirable. Stable model semantics is currently clearly the most widely accepted semantics for logic programs which have a stable model. We therefore believe that an extension of stable model semantics should preserve the meaning for those programs. On the other hand, many authors consider it a severe weakness of stable model semantics that not all normal logic programs have stable models. Our semantics overcomes this weakness. Finally, there has been a great amount of recent work trying to extend the expressiveness of normal logic programs by weakening the restrictions on the syntactic form of the rule heads. It turns out to be a non-trivial task to adapt existing semantics to these generalized logic programs. It is therefore clearly an advantage if a simple semantics for normal programs can directly be applied to theses generalizations.

Using abductive frameworks to define a semantics for logic programs is not a new idea. Eshghi and Kowalski [EK89] were the first to investigate logic programs, in particular negation as failure, in terms of abduction. More recently, Kakas and Mancarella [KM90] and Dung [Dun91] have continued this line of research. We will in the rest of this section review these earlier approaches and discuss why we do not consider them entirely satisfactory.

Eshghi and Kowalski show that negation as failure in normal logic programs can be viewed as a special case of abduction. Their analysis is based on abductive frameworks of the form $< T, I, A >$ where T is a set of definite clauses, I a set of integrity

[1] For simplicity we consider only finite propositional logic programs, that is programs with finite Herbrand base, in this preliminary report. All definitions also apply to the general case.

constraints, and A a set of abducible predicates. Atomic ground formulas built from the symbols in A are called abducibles. A set of abducibles Δ is an abductive solution for q iff $T \cup \Delta \vdash q$ and $T \cup \Delta$ satisfies I.

Since the authors restrict T to definite clauses and the set of abducibles to atomic formulas they have to eliminate negation signs from logic programs to handle negation as failure in their framework. Given a normal logic program P they introduce for each predicate symbol p in P a new predicate symbol p^*. P is then transformed to a program P^* by replacing each negative occurrence of a predicate symbol p in the body of a rule by a positive occurrence of p^*. Additionally, integrity constraints of the form

$$\leftarrow p^*(x) \wedge p(x)$$

are used to make sure that not both $p^*(t)$ and $p(t)$ can be true at the same time. This, however, is still insufficient to capture the meaning of negation as failure in logic programs. Consider the following program P_1:

$$a \leftarrow \neg b$$
$$b \leftarrow \neg c$$

In the Eshghi/Kowalski approach P_1 is transformed to ${P^*}_1$:

$$a \leftarrow b^*$$
$$b \leftarrow c^*$$

Now a has an abductive solution $\{b^*\}$ if only integrity constraints of the form mentioned above are used, contrary to the standard interpretation of logic programs where only the second rule in the original program can be applied. Eshghi and Kowalski handle this problem by adding to the integrity constraints metalevel constraints of the form

$$Demo(T \cup \Delta, p^*(t)) \vee Demo(T \cup \Delta, p(t))$$

Such a disjunctive integrity constraint is satisfied iff $p(t)$ or $p^*(t)$ is provable from $T \cup \Delta$. In our example this leads to the exclusion of the abductive solution for a, since from ${P^*}_1 \cup \{b^*\}$ neither c^* nor c can be proven, in contradiction to the integrity constraints. The only abductive solution for ${P^*}_1$ is $\{c^*, a^*\}$.

Note that this solution corresponds to the single stable model of P_1, $\{b\}$. This is not incidental: Eshghi and Kowalski show that for every stable model of a program P there is a corresponding abductive solution for the transformed program P^* and vice versa. This one-to-one correspondence to stable models shows that the new abductive semantics does not give meaning to programs without stable models and thus inherits the weakness of stable model semantics: the existence for abductive solutions for normal programs is not guaranteed. Consider the program P_2

$$p \leftarrow \neg p$$

and its transform

$$p \leftarrow p^*$$

The introduction of the metalevel constraint leads to the non-existence of an abductive solution.

Kakas and Mancarella use a similar abductive framework as Eshghi and Kowalski to define a generalization of stable models. In their paper an abductive framework is a triple $< P, A, I >$ where P is a normal logic program, A a set of abducible predicates, and I a set of integrity constraints. Contrary to Eshghi/Kowalski they directly use the notion of a stable model in their definitions. A pre-generalized stable model of $< P, A, I >$ is a stable model of $P \cup \{p \leftarrow \mid p \in \Delta\}$, where Δ is an arbitrary set of abducibles. A generalized stable model is a pre-generalized stable model that implies all integrity constraints in I.

The authors then show how negation as failure can be treated through abduction. The approach is similar to Eshghi/Kowalski's but somewhat simpler: the metalevel constraints involving the Demo predicate are replaced by integrity constraints of the form

$$p(x) \vee p^*(x)$$

Obviously, the stable models of a normal program P are exactly the generalized stable models of the abductive framework $< P, \emptyset, \emptyset >$. From this it is immediate that the existence of generalized stable models is not guaranteed. Generalized stable models thus do not solve the problem of non-existence of stable models for normal programs.

Dung's abductive frameworks [Dun91] are equivalent to Esghi/Kowalski's, that is he requires the programs in frameworks to consist of definite clauses. To be able to handle normal programs he replaces predicate symbols p in negated literals by new symbols p^*, as in Eshghi/Kowalski's approach. He also uses integrity constraints of the form

$$\leftarrow p^*(x) \wedge p(x)$$

but no constraints corresponding to the metalevel constraints involving the predicate *Demo*. Atoms built from the new symbols become abducibles.

$S = P \cup H$ is a scenario of the abductive framework $< P, A, I >$ if H is a subset of the abducible atoms such that $P \cup H \cup I$ is consistent. Let $inout(S)$ denote the set of ground atoms provable from a scenario S. A set of abducible atoms E is a P-evidence for an atom p iff $P \cup E \vdash p$. An abducible $p^*(t)$ is S-acceptable iff for every P-evidence E of $p(t)$, $E \cup inout(S) \cup I$ is inconsistent. A scenario S is admissible if every abducible atom in S is S-acceptable. An admissible scenario is complete iff every abducible atom that is also S-acceptable is contained in S.

The complete scenarios define the semantics of a normal logic program. Dung was able to show that the set of complete scenarios forms a semi-lattice with respect to set inclusion. The maximal complete scenarios correspond to stable models, and the least complete scenario to the well-founded model.

The abductive framework we a re going to present in this paper is distinct from this earlier work in the following respects:

1. We do not restrict the abducibles to atoms. This has the advantage that we can operate on the original programs directly and do not have to use any kind of transformation of the programs. Moreover, this makes the use of integrity constraints unnecessary.

2. We actually consider the rules of a program as inference rules, not as clauses.

3. We apply a new simple maximality criterion that guarantees that the right sets of abducibles are chosen.

4. Our framework is simpler than the existing approaches and can, contrary to the mentioned approaches, be applied to generalized logic programs without further modification.

2 The abductive framework

In this section we will introduce our abductive semantics for logic programs. We will define the notion of an extension for a logic program. This terminology reflects the similarity to other work in nonmonotonic reasoning, in particular default logic.

Although our framework is general enough to cover generalized logic programs we will only be concerned with normal programs in this section. For expository reasons we postpone the discussion of generalized logic programs until Section 3. Let us first define what we mean by a normal program:

Definition 1 *A normal logic program P is a set of rules of the form*

$$a \leftarrow b_1, \ldots, b_n$$

where a is an atom and the b_i are literals.

The notion of consistency plays a predominant role in abduction. We therefore have to define its meaning in the context of a logic program:

Definition 2 *Let L be a set of literals, P a logic program. The closure of L under P, $C_P(L)$, is the smallest set such that*

1. $L \subseteq C_P(L)$,

2. *if $a \leftarrow b_1, \ldots, b_n \in P$ and $b_1, \ldots, b_n \in C_P(L)$*
   then $a \in C_P(L)$.

Definition 3 *Let L be a set of literals, P a logic program. L is P-consistent iff $C_P(L)$ is consistent.*

Definition 4 *Let P be a logic program. P is consistent iff $\emptyset$ is P-consistent.*

We use $NEG(P)$ to denote the set of negated atoms of a logic program P, i.e. $NEG(P) = \{\neg a \mid a \text{ is an atom appearing in } P\}$.

Similar to the earlier abductive treatments of logic programs we model negation as failure abductively. However, since we do not require abducibles to be atoms we do not need to transform programs but can directly use $NEG(P)$ as the set of abducibles. The main difficulty is that we cannot consider all maximally consistent subsets of $NEG(P)$ as representing the intended meaning of a program. This simple approach fails to capture the intuitions underlying logic programming as can be demonstrated by our program P_1. This program has been used earlier to illustrate a similar problem for the Esghi/Kowalski approach:

$$a \leftarrow \neg b$$
$$b \leftarrow \neg c$$

The standard reading of this program is that b is derivable and a underivable. However, there exists a maximal P-consistent subset of $NEG(P)$, namely $H_1 = \{\neg b\}$ that fails to capture this intuition. The closure of H_1 under P_1 contains a but not b. This clearly violates all of the standard semantics for logic programs, and H_1 should not be considered an acceptable set of hypotheses.

What then are the acceptable sets of hypotheses, or - in our terminology - the extension bases, that can be used to define the meaning of a logic program? It turns out that an idea used in [Kon92] for reasoning about simple causal systems can be applied to solve this problem. The reader may observe that among the two maximally consistent subsets of $NEG(P)$ in the above example, the wanted subset, $H_2 = \{\neg c, \neg a\}$ allows us to derive b, that is refutes the remaining hypothesis in $NEG(P)$. The unintended subset, on the other hand, does not refute the hypothesis $\neg c$. It turns out that, to capture the intuition behind logic programs, we have to maximize not just the set of hypotheses, but also the set of refuted hypotheses. This leads to the following definitions:

Definition 5 *Let P be a logic program and $H \subseteq NEG(P)$. The P-cover of H, $COV_P(H)$, is the set*

$$H \cup \{\neg a \in NEG(P) \mid a \in C_P(H)\}$$

Definition 6 *Let P be a logic program. $H \subseteq NEG(P)$ is an extension base of P iff*

1. *H is P-consistent,*
2. *there is no P-consistent set H' such that $COV_P(H) \subset COV_P(H')$.*

Definition 7 *Let P be a logic program. E is an extension of P iff $E = C_P(H)$ where H is an extension base of P.*

It is obvious that our example gives only rise to one extension, as intended, since $COV_P(H_1) = \{\neg a, \neg b\} \subset \{\neg a, \neg b, \neg c\} = COV_P(H_2)$. This extension coincides with the unique stable model of the program. This is not incidental. We can show that our semantics and stable model semantics coincide in cases where stable models exist.

Proposition 1 *Let P be a normal logic program for which a stable model exists. If M is a stable model of P, then the set $NEG(M) = \{\neg p \mid p \notin M\}$ is an extension base of P. Vice versa, if E is an extension base of P, then the set $POS(E) = \{p \mid p$ atom in $COV_P(E)\}$ is a stable model of P.*

Proof: For the proof we first recall the definition of a stable model. Let M be a model of a program P (interpreted as a set of implications). M is a stable model of P iff M is a minimal model of the reduct P_M defined as

$$P_M = \{a \leftarrow b_1 \ldots b_n \mid a \leftarrow b_1 \ldots b_n \neg c_1 \ldots \neg c_m \in P, c_i \notin M\}$$

1) Assume M is a stable model of P. We show that $NEG(M)$ is an extension base of P. Since P_M is a definite program and M is a minimal model of P_M, a positive literal p is contained in M iff it is contained in the closure of $NEG(M)$ under P. Thus $COV_P(NEG(M)) = NEG(P)$, that is there can be no $H' \subseteq NEG(P)$ with $COV_P(NEG(M)) \subset COV(H')$. Moreover, since M is a model of P (interpreted as a set of logical implications), $NEG(M)$ must also be consistent with P (interpreted as a set of rules). Therefore $NEG(M)$ is an extension base.

2) Let E be an extension base of P. We have to show that $M = POS(E)$ is a stable model of P. Since by assumption P has a stable model we know from 1) that there is an extension base H with $COV_P(H) = NEG(P)$, therefore $COV_P(E) = NEG(P)$, that is, for every atom p, if $p \notin M$ then $\neg p \in E$.

Furthermore, since P_M is a definite program we have that $p \in C_P(E)$ iff p is contained in the minimal model of P_M. Therefore $p \in M$ iff p is contained in the minimal model of P_M and hence M is a stable model of P. □

Obviously, the existence of extensions for normal logic programs is guaranteed since every such program must be consistent. This is achieved because, contrary to stable models, extensions do not have to contain either a or $\neg a$ for every atom a. Consider the following example:

$a \leftarrow \neg a$
$b \leftarrow \neg c$

This program has no stable model, yet it has an extension generated by the extension base $\{\neg c\}$.

It is not difficult to establish the relationship of this approach to default logic [Rei80]. We transform each rule of a logic program P to a corresponding default, that is we define

$$D_P = \{l_1 \wedge \ldots \wedge l_n : true/a \mid a \leftarrow l_1 \ldots l_n \in P\}$$

Moreover, we have to represent $NEG(P)$ as a set of prerequisite-free normal defaults, that is

$$D_{NEG} = \{true : \neg p/\neg p \mid \neg p \in NEG(P)\}$$

Now E is an extension of P iff it is a Reiter extension of a default theory

$$T = (D_P \cup D'_{NEG}, W)$$

where D'_{NEG} is a maximal subset of D_{NEG} such that T has an extension.

3 Generalized Logic Programs

In this section we will consider generalized logic programs, that is programs where arbitrary negations and disjunctions may appear in the head of a rule. It turns out that all we have to do is slightly generalize the notion of P-closure:

Definition 8 *Let F be a set of formulas. The P-closure of F, $C_P(F)$, is the smallest set such that*

1. $F \subseteq C_P(F)$,
2. $C_P(F)$ *is deductively closed,*
3. *if* $a \leftarrow b_1, \ldots, b_n \in P$, *and* $b_1, \ldots, b_n \in C_P(L)$
   then $a \in C_P(L)$.

We say F is P-consistent if $C_P(F)$ is consistent. All other definitions from the last section can now be applied without further changes to generalized logic programs. Here is an example involving disjunctions in the head of a rule:

$$a \vee b \leftarrow \neg c$$

We obtain two extension bases, $E_1 = \{\neg c, \neg b\}$ and $E_2 = \{\neg c, \neg a\}$. Both have the cover $NEG(P)$. Note that $\{\neg a, \neg b\}$ is not an extension base since its cover does not contain $\neg c$.

The following slight modification of the last example involves a negation in the head:

$$a \vee \neg b \leftarrow \neg c$$

Now there is only one extension base, namely $\{\neg a, \neg b, \neg c\}$.

Generalized programs can be used to implement many of the standard default reasoning examples. Here is a bird example:

$fly \leftarrow bird, \neg ab_1$
$\neg fly \leftarrow penguin, \neg ab_2$
$ab_1 \leftarrow \neg ab_2$
$penguin$
$bird$

We obtain one extension from the extension base $\{\neg ab_2, \neg fly\}$. The P-cover of this extension base is $NEG(P)$. Note that the set of abducibles $\{\neg ab_1\}$ is not an extension base as its P-cover does not contain $\neg ab_2$. As intended the more specific rule gets priority.

It should be noted that the introduction of negation in the heads of rules leads to a situation where the existence of extensions can no longer be guaranteed for all programs. The reason is that the programs themselves may become inconsistent. Recall that a program P is inconsistent if $C_P(\emptyset)$ is (classically) inconsistent. It is not difficult to prove the following lemma: [2]

Lemma 1 *Let P be a logic program. P has an extension iff P is consistent.*

Obviously, all programs where the negation sign (and $\bot$) does not appear in the head of a rule are consistent and therefore have at least one extension. This includes normal logic programs.

Sometimes it is convenient and useful to restrict the set of abducibles to a proper subset of $NEG(P)$. A logic program then consists of a set of rules P together with a set Hyp of negated literals representing the atoms assumed to be false by default. The P-cover of a set of abducibles $H \subseteq Hyp$ is, as before, the set of abducibles which are either assumed or refuted.

For instance, in the bird example we get the desired results if we restrict the abducibles to $Hyp = \{\neg ab_1, \neg ab_2\}$. Again we get one extension containing $\neg fly$. The extension base now is $\{\neg ab_2\}$. Note that the cover of this extension base is Hyp, whereas the cover of $H_2 = \{\neg ab_1\}$ does not contain $\neg ab_2$. H_2 therefore is no extension base.

There is also no reason why we should not sometimes let positive information, that is unnegated atoms, or even arbitrary formulas be contained in the set of abducibles. This gives us the possibility to generalize "negation as failure to derive" to "assertion as failure to refute". Assume we represent the bird example in the following, equivalent way

$fly \leftarrow bird, normal_1$
$\neg fly \leftarrow penguin, normal_2$

[2] The proof is easy since, as mentioned in the beginning, we only consider propositional programs with finite Herbrand base in this report. The author conjectures that this result carries over to the general case, but was so far unable to prove the impossibility of cases where infinite ascending chains $COV_P(H_1) \subset COV_P(H_2) \subset \ldots$ lead to non-existence of extensions.

$\neg normal_1 \leftarrow normal_2$
$penguin$
$bird$

Letting $Hyp = \{normal_1, normal_2\}$ obviously yields results which are equivalent to those of our original representation using ab-predicates.

Another interesting extension of this approach are prioritized logic programs. We may introduce explicit priorities among the hypotheses, e.g. in the style of preferred subtheories [Bre89]. The set of assumables Hyp can be divided into preference levels $H_1, H_2, \ldots$. An extension base E_1 is preferred to an extension base E_2 iff there is an i such that

1. $E_1 \cap (H_1 \cup \ldots \cup H_{i-1}) = E_2 \cap (H_1 \cup \ldots \cup H_{i-1})$, and
2. $E_2 \cap H_i \subset E_1 \cap H_i$.

Here is an example

$Pac \leftarrow Quaker, \neg ab_1$
$\neg Pac \leftarrow Rep, \neg ab_2$
$Quaker$
Rep

Let $Hyp = \{ab_1, ab_2\}$ and assume we want to give the Quaker rule priority. This can be done by splitting Hyp to $H_1 = \{\neg ab_1\}$ and $H_2 = \{\neg ab_2\}$. There are two extension bases, H_1 and H_2. It is easy to see that, according to our definition, the first one is preferred over the second one.

Remark: In this example this is the same as adding $ab_2 \leftarrow \neg ab_1$ to the program, that is we have a choice whether we want to represent priorities explicitly using an ordering on Hyp, or via additional rules using the available implicit prioritization. We suspect that explicit orderings make programs often more readable. Note that in case of a conflict between explicit and implicit priorities the implicit ones win since only extension bases are compared in our definition of preferred extension bases, and these respect the implicit priorities.

All of these possible further generalizations are easily accomodated in our framework.

4 Two types of negation?

In a recent paper [GL90] Gelfond and Lifschitz introduce logic programs with two different negation symbols, a "strong" negation denoted by $\neg$, and a "weak" negation denoted by $\sim$.

The use of two different negation symbols in the Gelfond/Lifschitz approach does not seem entirely convincing. First of all, two symbols are not necessary, as shown

by the authors. Strong negation can be eliminated by introducing p^* predicates representing the negation of the predicate p. Secondly, the difference between strong and weak negation seems to be more a technical than a semantical one. There is no clear semantical sense in which $\neg p$ can be considered "stronger" than $\sim p$. The authors even recommend using rules of the form

$$\neg p \leftarrow \sim p$$

which, in a usual reading of the term "stronger", make $\sim p$ stronger than $\neg p$. The only difference between the two negation signs which is independent of the program they appear in is the following: in order to derive $\neg p$ at least one rule must have been applied, which is not true for $\sim p$. All other differences depend on how a programmer uses the two symbols in his program.

To illustrate these difficulties let us consider McCarthy's train example. The authors use this example to motivate their approach:

$$cross \leftarrow \neg train$$

The intuition behind using the "strong" negation here is that *cross* should only be derivable if we are sure that there is no train, that is if our belief that there is no train does not merely depend on assumptions. But using $\neg train$ in the body of the rule certainly does not guarantee this. The derivation of $\neg train$ may depend on many assumptions, e.g. we may have a rule

$$\neg train \leftarrow \sim noise$$

or, as mentioned before, there may even be a rule

$$\neg train \leftarrow \sim train$$

It entirely depends on the program in which the rule appears whether $\neg train$ really is as strong as we want in our example.

There does also not seem to be a good reason why one would like to distinguish between strong and weak negative information, but not between strong and weak positive information. Also positive information that depends on assumptions may be considered weak. In particular if we admit non-negated assumables one might want to treat positive and negative information in a more symmetrical way. This could be done by introducing other new predicate symbols.

We therefore believe that the distinction between two negations should not be made part of the logic. This only hides the programmer's responsibility for the meaning of the negation symbols in a program. The Gelfond/Lifschitz translatability result shows that, fortunately, there is no need to extend the logic even if someone insists in such a distinction.

References

[Bre89] Brewka, G., Preferred Subtheories: An Extended Framework for Default Reasoning, Proc. IJCAI 89, Detroit, 1989

[Dun91] Dung, P.M., Negations as Hypotheses: An Abductive Foundation for Logic Programming, Proc. 8th Int. Conference on Logic Programming, Paris, 1991

[EK89] Eshghi, K., Kowalski, R.A., Abduction Compared with Negation by Failure, Proc. 6th Int. Conference on Logic Programming, 1989

[GL88] Gelfond, M., Lifschitz. V., The Stable Model Semantics for Logic Programming, Proc. 5th Int. Conference on Logic Programming, 1998

[GL90] Gelfond, M., Lifschitz. V., Logic Programs with Classical Negation, Proc. 7th Int. Conference on Logic Programming, 1990

[KM90] Kakas, A.C., Mancarella, P., Generalized Stable Models: A Semantics for Abduction, Proc. 9th European Conference on Artificial Intelligence, Stockholm, 1990

[Kon92] Konolige, K., A General Theory of Default Reasoning in Causal Domains, Proc. 4th Int. Workshop on Nonmonotonic Reasoning, Plymouth, VT, 1992

[Rei80] Reiter, R., A Logic for Default Reasoning, Artificial Intelligence 13, 1980