Soundness and Completeness of a Logic Programming Approach to Default Logic

Grigoris Antoniou
Elmar Langetepe
University of Osnabrueck, FB 6
49069 Osnabrueck, Germany
ga@informatik.Uni-Osnabrueck.DE

Abstract

We present a method of representing some classes of default theories as normal logic programs. The main point is that the standard semantics (i.e. SLDNF-resolution) computes answer substitutions that correspond exactly to the extensions of the represented default theory. We explain the steps of constructing a logic program LogProg(P,D) from a given default theory (P,D), and present the proof ideas of the soundness and completeness results for the approach.

Introduction

In last years much work is done to establish relationships between nonmonotonic reasoning and logic programming (Bidoit&Froidevaux 1991a, 1991b, Gelfond&Lifschitz 1988, 1991, Marek&Truszczynski 1989, Marek&Subrahmanian 1992, Pereira&Nerode 1993). This is usually done by defining a new semantics for logic programming (in particular of negation) and deriving a relationship to some nonmonotonic logic. In this paper we go the other way around: We maintain the classical standard semantics of logic programming (SLDNF-resolution) and try to translate a nonmonotonic logic into appropriate logic programs. In particular,

- we translate portions of Reiter's default logic (Reiter 1980) into normal logic programs, and
- prove that the translation LogProg(T) of a default theory T computes answer substitutions that exactly correspond to the extensions of T.

What are the benefits of such an approach?

- Standard semantics of logic programming is well analyzed and understood.
- Prolog is a powerful implementation of standard semantics.
- Therefore, we provide an implementational paradigm for nonmonotonic reasoning that might prove valuable in practice. For example, we might use parallel logic programming systems to achieve efficient reasoning systems.

Throughout the paper we assume familiarity with notation and basic notions of predicate logic and logic programming. In case of discomfort, please refer to (Lloyd 1987, Sperschneider&Antoniou 1991).

Basics of default logic

A *default* δ is a string $\varphi:\psi_1,\ldots,\psi_n/\chi$ with closed first-order formulas $\varphi,\psi_1,\ldots,\psi_n$ and χ (n>0). We call φ the *prerequisite*, $\psi_1,\ldots,\psi_n$ the *justifications*, and χ the *consequent* of δ. A *default schema* is a string of the form $\varphi:\psi_1,\ldots,\psi_n/\chi$ with arbitrary formulas. Such a schema defines a set of defaults, namely the set of all ground instances $\varphi\sigma:\psi_1\sigma,\ldots,\psi_n\sigma/\chi\sigma$ of $\varphi:\psi_1,\ldots,\psi_n/\chi$, where σ is an arbitrary ground substitution.

A *default theory* T is a pair (W,D) consisting of a set of closed formulas W (the set of truths) and a denumerable set of defaults D. The default set D may be defined using default schemata.

Let $\delta=\varphi:\psi_1,\ldots,\psi_n/\chi$ be a default, and E and F sets of formulas. We say that δ *is applicable to* F *with respect to belief set* E iff $\varphi\in$F, and $\neg\psi_1\notin$E,...,$\neg\psi_n\notin$E. F *is closed under* D *with respect to* E iff, for every default $\varphi:\psi_1,\ldots,\psi_n/\chi$ in D that is applicable to F with respect to belief set E, its consequent χ is also contained in F.

Given a default theory T=(W,D) and a set of closed formulas E, let $\Lambda_T(E)$ be the least set of closed formulas that contains W, is closed under logical conclusion and closed under D with respect to E.

A set of closed formulas E is called an *extension* of T iff $\Lambda_T(E)$=E.

In (Antoniou&Langetepe 1993)] we provided an operational characterization of extensions. Here we will show that it can be the starting point for implementational issues.

*1. **Definition** Let T=(W,D) be a default theory and* $\Pi=(\delta_0,\delta_1,\delta_2,\ldots)$ *a finite or infinite sequence of defaults from D not containing any repetitions (modelling an application order of defaults from D). We denote by* $\Pi[k]$ *the initial segment of* Π *of length k, provided the length of* Π *is at least k. Then we define the following concepts:*

- *In(*Π*) is Th(M), where M contains the formulas of W and all consequents of defaults occurring in* Π.
- *Out(*Π*) is the set of negations of all justifications of defaults occurring in* Π.
- Π *is called a **process** of T iff* δ_k *is applicable to In(*$\Pi[k]$*) w.r.t. belief set In(*$\Pi[k]$*), for every k such that* δ_k *occurs in* Π.
- Π *is called a **successful process** of T iff In(*Π*)*$\cap$*Out(*Π*)=*$\emptyset$*, otherwise it is called a **failed process**.*
- Π *is a **closed process** of T iff every* $\delta\in D$ *which is applicable to In(*Π*) with respect to belief set In(*Π*) already occurs in* Π.

In(Π) collects all formulas in which we believe after application of the defaults in Π, while Out(Π) consists of all those formulas which we should avoid to believe for the sake of consistency. The following result (for a proof see (Antoniou&Sperschneider 1993)) states the relationship between default logic extensions and processes.

*2. **Theorem** Let T=(W,D) be a default theory. If* Π *is a closed successful process of T, then In(*Π*) is an extension of T. Conversely, for every extension E of T there exists a closed, successful process* Π *of T with E=In(*Π*).*

Informal outline

Our approach of representing default theories by logic programs does not apply to arbitrary default theories, but to a subset thereof we call *Horn default theories*. The restrictions are as follows:

1. The truths of the default theories are given in Horn logic.
2. Each default has exactly one justification.
3. The set of defaults is finite.
4. The prerequisite, justification, and consequent of each default is a positive or negative ground literal. In this paper we further restrict attention to propositional literals, but only to keep the following sections technically simpler.

Restriction 1 is clear as we plan to use standard logic programming for deduction, while restriction 2 is only used for the sake of convenience and simplicity. 3 and 4 are restrictive, and we are working on weakening these conditions (for example by admitting conjunction). Nevertheless, Horn default theories are powerful enough to include say taxonomic default theories (Froidevaux 1986).

The main ideas of the translation of Horn default theories into normal logic programs are the following:

- Use negation as failure to model nonmonotonicity.
- Enumerate the defaults and represent a current process Π by the numbers of the defaults in Π.
- Replace negative literals $\neg p$ appearing in defaults by new predicates $\underline{p}$.

In the next section we show how negative literals can be treated using new predicates. Then we put all ideas together and present the entire translation.

The logical basis of treating negative literals

As negative literals may be used in defaults, we shall have to test derivability or nonderivability of negative literals from the knowledge base built at some stage (i.e. the In-set of the process built so far). Consider as a simple example the set of Horn formulas P = {p←q, ←p}; ¬q obviously follows from P. If we use new predicate symbols for negated atoms, we can translate P into the logic program P' = {p←q, $\underline{p}$←}. Unfortunately, there is no SLD-refutation of P'∪{$\underline{q}$} as should be. What is obviously missing is application of the rule p←q via contraposition, i.e. $\underline{q}$←$\underline{p}$. This gives rise to the following definition.

*3. **Definition** For a definite logic program P define* $\underline{P}$ *as* $\{\underline{B_i}\leftarrow\underline{A},B_1,\ldots,B_{i-1},B_{i+1},\ldots,B_n \mid A\leftarrow B_1,\ldots,B_n\in P,\ n\geq 1,\ i\in\{1,\ldots,n\}\}$.

Usage of P∪$\underline{P}$ is still insufficient as shown by the following example. Let P be {p←q,r , q←r, ←p}. Obviously, ¬r follows from P, but there is no SLD-refutation of P∪$\underline{P}$∪{←$\underline{r}$} = {p←q,r , $\underline{q}$←$\underline{p}$,r , $\underline{r}$←$\underline{p}$,q , q←r , $\underline{r}$←$\underline{q}$, ←p , ←$\underline{r}$}. Reflection on this example shows that missing is a fact r← that is needed in the

SLD-refutation. This means that when testing derivability of some $\underline{r}$ from $P \cup \underline{P}$, we must (temporarily) add an additional fact $r\leftarrow$ to the knowledge base. It can be shown that this approach is sound and complete; for a proof, see (Langetepe 1994).

4. Theorem *Let P be a definite logic program, A, $G_1,\ldots,G_k$ positive literals, and $P\cup\{\leftarrow G_1,\ldots,\leftarrow G_k\}$ consistent.*

(a) $P\cup\{\leftarrow G_1,\ldots,\leftarrow G_k\} \models \exists(\neg A) \Leftrightarrow$

$P\cup\underline{P}\cup\{\underline{G}_1\leftarrow,\ldots,\underline{G}_k\leftarrow\}\cup\{A\leftarrow\} \models \exists(\underline{A})$

(b) $P\cup\{\leftarrow G_1,\ldots,\leftarrow G_k\} \models \exists(A) \Leftrightarrow$

$P\cup\underline{P}\cup\{\underline{G}_1\leftarrow,\ldots,\underline{G}_k\leftarrow\} \models \exists(A)$

The logic program LogProg(P,D)

Given a Horn default theory T=(P,D), we construct a corresponding logic program LogProg(P,D).

0. step: Enumerate D

Enumerate the (finitely many) defaults in D: Let D be $\{\delta_0,\ldots,\delta_n\}$.

1. step: Build P_L

As stated before, we use a list to represent the defaults applied so far. This means that the consequents of the defaults in the process built so far are available in the knowledge base. Obviously, the truths of the default theory (i.e. P) are available at any stage. Therefore: For each program clause $p\leftarrow p_1,\ldots,p_k$ in P build the clause

$$p(L)\leftarrow p_1(L),\ldots,p_k(L).$$

2. step: Build $\underline{P}_L$

In the previous section we saw that $\underline{P}$ is necessary for the derivation of negative literals. Equip the rules in $\underline{P}$ with a list L as seen in the 1. step.

3. step: Build Supplement(D)

- For each default δ_i in D with consequent r add the clause

  $$r(L)\leftarrow member(i,L).$$

  For each default δ_i in D with consequent $\neg r$ add the clause

  $$\underline{r}(L)\leftarrow member(i,L).$$

  The meaning is what we have already stated before: If i is member of L then δ_i has been applied, so its consequent should be in the current knowledge base (determined by L).

- If the prerequisite of δ_i is $\neg p$ then add

  $$p(L)\leftarrow member([pre,i],L).$$

  The meaning gets clear if we think back to Theorem 4. There we saw that in order to test derivability of a negative literal $\neg p$ (as in the test of applicability of default δ_i), we must *temporarily* add p to the current knowledge base. Exactly this is the meaning of the added rule: we are adding p but in a distinguished way (indicated by *pre*), so that p is not part of the current knowledge base but is only used in the derivability test of $\underline{p}$.

- If the justification of δ_i is a positive literal q then add

  $$q(L)\leftarrow member([conscheck,i],L).$$

  The explanation of this rule is the same as above, if we recall that consistency check means to try to derive the negation of the justification, in our case $\neg q$, a negative literal.

4. step: Build program Member

$$member(X,[X|L]) \leftarrow$$

$$member(X,[Y|L]) \leftarrow member(X,L).$$

5. step: The control structure Process(D)

The function of the control structure is to systematically build processes by applying defaults and backtracking, if necessary (i.e. if a process is failed or if another extension is sought).

Let $i\in\{0,\ldots,n\}$ (recall that $D = \{\delta_0,\ldots,\delta_n\}$) and δ_i be the default p_i:B/C. Define $\underline{q}_i$ as $\underline{q}$ if B is a positive atom q, and as q if B is a negative atom $\neg q$. Similarly, define $\underline{r}_i$ as $\underline{r}$ if C is a positive atom r, and as r if B is a negative atom $\neg r$. Add to the logic program the rule *proc(i)*:

```
process(Lold,L) ← consistent(Lold),
                  not member(i,Lold),
                  p_i([[pre,i] | Lold]),
                  not q_i([[conscheck,i] | Lold]),
                  process([i | Lold],L).
```

Now we add some additional rules, namely the termination case for the predicate process, and the implementation of the predicates consistent, closed, and successful.

```
process(L,L) ← consistent(L), closed(L),
               successful(L)
closed(L) ← not ( not member(0,L),
                  p_0([[pre,0] | L]),
```

not $\underline{q}_0$([[conscheck,0] | L])),

. . .

not (not member(n,L),
p_n([[pre,n] | L]),
not $\underline{q}_n$([[conscheck,n] | L]))

successful(L) ←
not (member(0,L), $\underline{q}_0$([[conscheck,0] | L])),

. . .

not (member(n,L), $\underline{q}_n$([[conscheck,n] | L])),

consistent([ ]) ←
consistent([i | Lold]) ← not $\underline{r}_i$([i | Lold]).

(the last rule, of course, for each i∈ {0,...,n}).

6. step: LogProg(P,D)

Define LogProg(P,D) as

$P_L \cup \underline{P}_L \cup$Supplement(D)∪Member∪Process(D).

Explanations and examples of this approach can be found in (Antoniou&Langetepe 1994).

Remark on a Prolog implementation

When implementing the normal logic program LogProg(P,D) in Prolog, some obvious modifications must be carried out. These include a more efficient implementation of member, and usage of one Prolog rule instead of the finite collection of rules *proc(i)* for the predicate process and corresponding rules in the definition of closed and consistent. The reason we use separate rules in LogProg(P,D) is of technical nature in order to derive soundness and completeness; it is easily seen that the Prolog implementation with one rule would not lead to a normal logic program (predicates p_i and $\underline{q}_i$ would dynamically depend on the default i chosen).

The main result

Our approach is sound and complete - the entire proof is given in (Langetepe 1994); in the next section we present the main steps). Note that the additional condition in the completeness result (hierarchical program P) has nothing to do with our approach, but is the usual restriction in the completeness of SLDNF-resolution.

5. Theorem *Let (P,D) be a Horn default theory.*

(a) If there exists an SLDNF-refutation of

$LogProg(P,D) \cup \{\leftarrow process([\,],L)\}$,

then the computed answer substitution has the form

$\{L/[i_0,\ldots,i_k]\}$

with $i_j \in \{0,\ldots,n\}$, *and*

$\Pi = (\delta_{i_k},\ldots,\delta_{i_0})$

is a closed, successful process of (P,D), i.e. In(Π) is an extension of (P,D).

(b) If P is hierarchical and $\Pi = (\delta_{i_0},\ldots,\delta_{i_k})$ *is a closed, successful process of (P,D), then there exists an SLDNF-refutation of*

$LogProg(P,D) \cup \{\leftarrow process([\,],L)\}$

with computed answer substitution $\{L/[i_k,\ldots,i_0]\}$.

The proof outline

Both soundness and completeness of our approach are shown using the operational semantics of LogProg(P,D), i.e. by analyzing the structure of an SLDNF-refutation of LogProg(P,D)∪{←process([ ],L)}. Because of space limitations we only give the main steps and lemmata of the proof.

Soundness

First we show that representing inclusion of formulas in the current knowledge base as done in LogProg(P,D) works in the intended way.

6. Lemma *Let P be a definite logic program,* $p_i(t_i)$ *(i = 1 ,2 ,3) ground atoms, A an atom, L' a ground list, and* t_i' *(i = 1, 2, 3) ground terms such that* $t_1', t_2' \in L'$ *and* $t_3' \notin L'$. *Assume that predicate* member *does not occur in P or in A. Then: An SLDNF-refutation of*

$P \cup \{p_1(t_1)\leftarrow\} \cup \{\leftarrow A\}$

exists iff there is an SLDNF-refutation of

$P_L \cup Member \cup \{p_i(t_i)\leftarrow member(t_i',L') \mid i = 1, 3\} \cup \{\leftarrow A_L'\}$.

The ground atoms $p_i(t_i)$ cover the three possible cases: $p_1(t_1)$ is included in the knowledge base, $p_2(t_2)$ and $p_3(t_3)$ are not: the first one because of the lacking corresponding rule, the latter because $t_3' \notin L'$. Lemma 6 can be extended to an arbitrary finite number of ground atoms and to arbitrary goals (in Horn logic). The following lemma gives some structure properties of SLDNF-refutations of LogProg(P,D)∪{←process([ ],L)}.

7. Lemma *An SLDNF-refutation of LogProg(P,D)∪{←process([ ],L)} has the following properties:*

(a) There are k resolution steps with side clauses proc(i), and then one step with side clause with

head process(L,L). No other rule from Process(D) is used afterwards.

(b) Each intermediate goal G includes at most one literal with predicate **process**; *the literal has the form process(L_1,L_2) with a ground list L_1 and a variable L_2 occurring only once in G.*

Further analysis the SLDNF-refutation of LogProg(P,D)∪{←process([ ],L)} using the result above leads to the following

8. Lemma *Suppose there is an SLDNF-refutation of LogProg(P,D)∪{←process([ ],L)} with computed answer substitution μ. Then:*

(a) $\mu = \{L/[i_0,\ldots,i_k]\}$ *with* $i_j \in \{0,\ldots,n\}$

(b) For m = k, k-1,...,0 there are SLDNF-refutations of

LogProg(P,D)∪{← not $\underline{r}_{i_{k-m}}([i_{k-m},\ldots,i_0])$*}*

LogProg(P,D)∪{← not member(i_{k-m},$[i_{k-m-1},\ldots,i_0]$*)}*

LogProg(P,D)∪

{← $p_{i_{k-m+1}}([\,[pre,i_{k-m}]\backslash[i_{k-m-1},\ldots,i_0]\,])$ *}*

LogProg(P,D)∪

{← not $\underline{q}_{i_{k-m}}([[conscheck,i_{k-m}]\backslash[i_{k-m-1},\ldots,i_0]])$*}.*

(c) For each $i \in \{0,\ldots,n\}$ *exists a finitely failed SLDNF-tree of*

LogProg(P,D)∪{← not member(i,$[i_k,\ldots,i_0]$*)} or of*

LogProg(P,D)∪{← $p_i([\,[pre,i]\backslash[i_k,\ldots,i_0]\,])$ *} or of*

LogProg(P,D)∪{← not $\underline{q}_i([[conscheck,i]\backslash[i_k,\ldots,i_0]])$*}.*

(d) For each $i \in \{0,\ldots,n\}$ *exists a finitely failed SLDNF-tree of*

LogProg(P,D)∪{←member(i,$[i_k,\ldots,i_0]$*)} or of*

LogProg(P,D)∪{← $\underline{q}_i([[conscheck,i]\backslash[i_k,\ldots,i_0]])$*}.*

Using this lemma and soundness of SLDNF-resolution it is not difficult to show that indeed defaults $\delta_{i_k},\ldots,\delta_{i_0}$ may be applied in this order and form a closed, successful process Π of (P,D). This shows that the computed answer substitution $\{L/[i_0,\ldots,i_k]\}$ determines the generating defaults of the extension In(Π) of (P,D).

Completeness

9. Lemma *Let P be a hierarchical logic program. Then, P∪*$\underline{P}$ *is also hierarchical.*

Proof scetch Let m := max{level(p) | p predicate symbol in P}. Define

level($\underline{p}$) := 2m - level(p) + 1.

With this definition it is easy to check that P∪$\underline{P}$ is hierarchical. Now suppose $\Pi = (\delta_{i_0},\ldots,\delta_{i_k})$ is a closed, successful process of (P,D) (i.e. In(Π) is an extension of the default theory (P,D)). By definition of processes we have for all m = k, k-1,...,1

- $i_0,\ldots,i_k$ are pairwise disjoint
- $P\cup\{r_{i_s} \mid s = 0,\ldots,k\text{-}m\text{-}1\} \models p_{i_{k-m}}$
- $P\cup\{r_{i_s} \mid s = 0,\ldots,k\text{-}m\text{-}1\} \not\models \neg q_{i_{k-m}}$
- $P\cup\{r_{i_s} \mid s = 0,\ldots,k\text{-}m\text{-}1\}$ is consistent.

Applying the results of Theorem 4, completeness of SLDNF-resolution for hierarchical programs, and Lemma 6 we can show existence of

an SLDNF-derivation of ←process([i_k,...,i_0],L)
from LogProg(P,D)∪{←process([ ],L)} (*)

From the information that P is closed and successful we may show that there exist SLDNF-refutations of

LogProg(P,D)∪{←closed([i_k,...,i_0])}
LogProg(P,D)∪{←successful([i_k,...,i_0])} (**)
LogProg(P,D)∪{←consistent([i_k,...,i_0])}

Resolving ←process([i_k,...,i_0],L) (from (*)) with side clause

process(L',L') ← consistent(L'), closed(L'), successful(L')

using a most general unifier {L/[i_k,...,i_0], L'/[i_k,...,i_0]} and applying (**) we finally derive the empty clause. Altogether, we have constructed an SLDNF-refutation of LogProg(P,D)∪{←process([ ],L)} with computed answer substitution {L/[i_k,...,i_0]}.

Conclusion

We gave an automated translation of a class of default theories into normal logic programs. The main contribution of this paper is the proof (outline) of the result that the answer substitutions computed by the logic program via its standard semantics correspond exactly to the extensions of the default theory.

There is still much work to be done. First, we are working on weakening some restrictions to the default theories. We are thinking especially of admitting conjunction and investigating cases where infinite theories and extensions could be treated (using unification).

We are also planning to make an experimental

comparison with other methods of implementing default logic like truth maintenance systems, graphs, the system Theorist (Poole 1988) etc. One of the most promising ideas is to exploit the parallelism in the logic programs given in this paper and thus in providing an efficient, parallel implementation of (parts of) default logic.

Finally, we are planning to investigate the relevance of our translation of default theories into logic programming when using nonstandard semantics.

References

Antoniou, G., and Sperschneider, V. 1993. Computing Extensions of Nonmonotonic Logics. In Proceedings 4th Scandinavian Conference on Artificial Intelligence, IOS Press.

Antoniou, G., and Langetepe, E. 1993. A Process Model for Default Logic and its realization in Logic Programming. In Proceedings Portuguese Conference on Artificial Intelligence (EPIA-93), Springer LNAI.

Antoniou, G., and Langetepe, E. 1994. Translation of parts of default logic into normal logic programs with standard semantics. In Proceedings European Conference on Artificial Intelligence (submitted).

Bidoit, N., and Froidevaux, C. 1991a. General Logic Databases and Programs: Default Logic Semantics and Stratification. *Information and Computation* 91, 15-54.

Bidoit, N., and Froidevaux, C. 1991b. Negation by Default and Unstratifiable Logic programs. *Theoretical Computer Science* 78, 85-112.

Froidevaux, C. 1986. Taxonomic Default Theory. In Proceedings European Conference on Artificial Intelligence(ECAI-86).

Gelfond, M., and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In Proceedings 5th Int. Conference/Symposium on Logic Programming, 1070-1080.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9, 365-385.

Langetepe, E. 1994. Betrachtung einzelner Default-Logik Ansätze unter Verwendung eines operationalen Extensionsmodells. Masters Thesis, Fachbereich 6, Universität Osnabrück.

Lloyd, J.W. 1987. *Foundations of Logic Programming* 2. edition. Springer.

Marek, W., and Truszczynski, M. 1989. Stable Semantics for Logic Programs and Default Theories. In Proceedings North American Conference on Logic Programming, 243-256.

Marek, W., and Subrahmanian, V.S. 1992. The Relationship between Stable, Supported, Default and Autoepistemic Semantics for General Logic Programs. *Theoretical Computer Science* 103, 365-386.

Pereira, L.M., and Nerode, A. 1993. *Logic Programming and Non-monotonic Reasoning, Proceedings of the 2nd International Workshop*. MIT Press.

Poole, D. 1988. A Logical Framework for Default Reasoning. *Artificial Intelligence* 36.

Reiter, R. 1980. A Logic for Default Reasoning. *Artificial Intelligence* 13.

Sperschneider, V., and Antoniou, G. 1991. *Logic: A Foundation for Computer Science*. Addison-Wesley.