Dealing with Exceptions in Textual Databases*

Renwei Li[1] Veronica Dahl[2] Luis Moniz Pereira[1] Marius Scurtescu[2]

[1] Department of Computer Science
Universidade Nova de Lisboa
2825 Monte de Caparica
Portugal
{renwei,lmp}@di.fct.unl.pt

[2] Logic Programming Group
School of Computing Science
Simon Fraser University
Burnaby, B.C. V5A 1S6, Canada
{ veronica, mas}@cs.sfu.ca

Abstract

This paper addresses the semantic issue of exceptions in natural language texts. After observing that first-order logic is not adequate for semantic representations of natural language, we make use of extended logic programs for meaning representation and inference engine in natural language processing. We show how to generate semantics of texts through deductive parsing by using definite clause grammars with focus on meanings of exceptions. Given a textual database, its semantics can be produced as an extended logic program, and the query against the textual database is thus reduced to a query in an extended logic program. The non-monotonicity of extended logic programs makes our method to be especially applicable to incomplete textual databases, but it is also generally applicable to the analysis of natural language descriptions in view of conceptual modelling and to natural language interfaces for database querying. Generally speaking, it also provides a method for conflict resolution when the conflict arises from exceptions.

1 Introduction

Suppose we have a textual database D and a query Q. In order to evaluate Q against D, first of all we need to understand the meaning of D and Q. In the literature on natural language processing, first-order logic is often used as the semantic representation language for meanings of natural language texts. However, when we say something or write something, we may make some common-sense assumptions. For example, when we say *every bird flies*, we may not mean that *if B is a bird, B flies*. What we really mean is that *every bird* **normally** *flies*. This observation naturally leads to the choice of non-monotonic reasoning systems for meaning representations of natural language. In fact, examples from natural languages are often used as motivations for the development of non-monotonic reasoning systems, but little effort has been made towards automating the generation of default theories from natural language texts. On the other hand, there have been several competing formalisms for non-monotonic reasoning [11, 5, 6]. Recent developments have shown that logic programming is an adequate tool for nonmonotonic reasoning [9, 7]. In this paper we will make use of extended logic programs with well-founded model semantics for meaning representations of natural language.

Technically, we will use Definite Clause Grammars to parse English texts. If a text is accepted as a valid sentence, its semantics (an extended logic program) is automatically produced. A natural language query itself is also parsed by using Definite Clause Grammar. Evaluating a natural language query against a textual database amounts to evaluating a query in an extended logic program.

*published in: P. McFetridge (ed.), *Proc. of NLDB'97*, 1997, pp. 126–135

The rest of this paper is organized as follows. In Section 2 we describe extended logic programming which will be used as the underlying inference engine of our system and the meaning representation language. In Section 3 we briefly review Definite Clause Grammars (DCGs) and their application to syntactical parsing and semantics generation. In Section 4 we discuss exceptions and how to generate their meanings. In Section 5 we give a complete example to show how to generate meanings of exceptions. In Section 6 we briefly introduce a textual database query system. In Section 7 we conclude this paper with a few remarks.

2 Extended Logic Programs

The extended logic programs will be used to represent the meaning of natural language statements. An extended logic program consists of rules of the form:

$$L \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n.\ (0 \leq m \leq n)$$

where the symbol *not* denotes the negation-as-failure operator, each of L_1 and L_i is either an atom or an atom preceded by explicit negation $\neg$. Atoms and atoms preceded by explicit negation $\neg$ are called objective literals. The set of all ground objective literals of an extended logic program P is called the extended Herbrand base of P and denoted by $\mathcal{H}(P)$. By $not\ \{a_1, \ldots, a_n, \ldots\}$ we mean $\{not\ a_1, \ldots, not\ a_n, \ldots\}$. An interpretation I of an extended logic program P is denoted by $T \cup not\ F$, where T and F are two subsets of $\mathcal{H}(P)$. Objective literals in T are said to be *true* in I, objective literals in F *false by default* in I, and objective literals in $\mathcal{H}(P) - I$ *undefined* in I.

If the explicit negation $\neg$ does not appear in an extended logic program P, then it collapses to a normal logic program; if the negation-as-operator does not appear, it collapses to a definite logic program; if the body of every rule is empty, it collapses to a relational database. For many interesting domains, relational databases are too weak and extended logic programs are not necessary, but locally stratified logic programs [10] are sufficient[1]. Anyway extended logic programs are sufficient to subsume many other knowledge bases.

The semantics of an extended logic program P is defined to be the well-founded semantics of P incorporated with a coherence principle [3]. The coherence principle relates two forms of negation: For any interpretation $I = T \cup not\ F$ and any objective literal L_1, if $\neg L \in T$ then $L \in F$. We start with the definition of Gelfond–Lifschitz module operator $\frac{P}{I}$.

Definition 2.1 (Modulo Operator $\frac{P}{I}$) Let P be an extended logic program, I an interpretation. Nondeterministically perform the following three operations: (i) Remove from P all rules containing a default literal $L = not\ A$ such that $A \in I$; (ii) Remove from P all rules containing an objective literal L_1 in the body such that $\neg L \in I$; (iii) Remove from all remaining rules of P their default literals $L = not\ A$ such that $not\ A \in I$. $\frac{P}{I}$, P modulo I, is the program obtained from the above result by replacing all the remaining default literals with a new proposition symbol $\mathbf{u}$.

Obviously, the resulting program $\frac{P}{I}$ is non–negative, i.e. no default literals appear in it, thus there is a unique Fitting-least 3-valued model, $least(\frac{P}{I})$, which is a generalization of the van Emden-Kowalski least model operator for definite logic programs. Note that $\frac{P}{I}$ is not unique because of the nondeterministic operations.

Definition 2.2 We define $least(P)$, where P is a non–negative program, as the set of literals $T \cup not\ F$ obtained as follows: (i) Let P' be the non-negative program obtained by replacing in P every negative objective literal $\neg L$ with a new atomic symbol, say $'\neg\_L'$; (ii) Let $T' \cup not\ F'$ be the least 3–valued model of P'; (iii) $T \cup not\ F$ is obtained from $T' \cup not\ F'$ by reversing the replacements above.

[1] Note that both stratified logic programs and acyclic logic programs are instances of locally-stratified logic programs. What is important is that the stable models and well-founded models of locally stratified logic programs coincide.

Although $least(P)$ uniquely exists for every non-negative program P, it might not satisfy the coherence principle, and hence it isn't always an interpretation.

Definition 2.3 (Coh operator) Let $QI = QT \cup not\ QF$ be a set of literals. We define $Coh(QI)$ to be the interpretation $T \cup not\ F$ such that $T = QT$ and $F = QF \cup \{\neg L \mid L \in T\}$.

Definition 2.4 (Φ operator) Let P be an extended logic program, I an interpretation. We define $\Phi_P(I) = \bigcup Coh(least(P_i))$, where P_i is some $\frac{P}{I}$.

It can be shown that the operator Φ_P for any program P is monotonic, and hence there is a least fixpoint $lfp(\Phi_P)$.

Proposition 2.5 *There is a least fixpoint of Φ_P for any extended logic program.*

Then, the well-founded semantics of P with explicit negation, denoted by WFSX(P), is defined to be the least fixpoint of Φ_P. For lack of space we will not go into deeper discussions on WFSX semantics. Many interesting results are summarized in the monograph [3]. In particular, the following result from [3] acts as the theoretical basis for the knowledge base manager.

Proposition 2.6 Let L_1 be any given objective literal and P any extended logic program. There is a decision procedure taking L_1 and P as inputs such that (i) if $L \in WFSX(P)$ then the decision procedure will always terminate in a finite time and returns the affirmative result, and (ii) if the decision procedure terminates and returns the affirmative result, then $L \in WFSX(P)$.

3 Grammar and Semantic Composition

As is known, a grammar and parser can be used not only to produce syntactic structure, but also to produce semantic representation [1]. For simplicity we will only make use of Definite Clause Grammar (DCG) for both syntactic and semantic analysis.

Invisibly to the user, the DCG compiler adds two arguments to each non terminal. The first argument records the input string represented as a list, and the second argument records a new string represented as a list, representing what is left of the input string after the symbol in question has consumed what it needs from it. In order to produce semantic representation, additional arguments must be added to a non-terminal in a DCG grammar. When a sentence is being parsed, all arguments will be unified with their corresponding semantic denotations, and after it is successfully parsed, the semantics will then be produced.

For example, let's consider a language consisting of only the following sentence E_1:

Every bird flies.

To parse its syntactic structure, the following DCG is sufficient:

```
s  --> np, vp.
np --> d, n.
vp --> vi.
d  --> [every].
n  --> [bird].
vi --> [flies].
```

Loading such a DCG into a prolog system such as SICSTUS, we can then determine whether a string S is a sentence of the above language by issuing a query[2]:

```
:- s(S, []).
```

[2] The reader unfamiliar with DCG is encouraged to run the above program in his Prolog system.

In order to produce its semantics, an additional argument may be added to the non-terminals in the above grammar[3]:

```
:- op(600,xfy,=>).

s(Sem) --> np((X^Sco)^Sem), vp(X^Sco).

np((X^Sco)^Sem) --> d((X^Res)^(X^Sco)^Sem), n(X^Res).

vp(X^P) --> vi(X^P).

d((X^P1)^(X^P2)^(all(X): (P1 => P2))) --> [every].

n(X^bird(X)) --> [bird].

vi(X^fly(X)) --> [flies].

go(Sent, Sem) :- string(Sent), s(Sem, Sent, []).

string([every,bird,flies]).
```

To generate the semantics, we may issue a query like this:

```
:- s(S,[every,bird,flies],[]).
```

the outcome of which is:

```
S = all(A,bird(A)=>fly(A))
```

The above semantic representation corresponds to the following more usual first-order sentence:

$$(\forall x)(bird(s) \supset fly(s))$$

4 Exceptions

Exceptions in natural language can be generally classified into two types: explicit and implicit exceptions. An explicit exception covers sentence patterns like this: sentence-1 unless sentence-2. It is relatively easy to generate the semantics of sentences with explicit exceptions. In this paper will will focus on implicit exceptions. Implicit exceptions arise when the exception follows from our common sense knowledge, rather than being explicitly stated.

Let's continue the above example, and consider a language consisting only of the following text:

Every bird flies. Tweety is a bird.

This example is often used in the literature on logic programming and non-monotonic reasoning. We now study it from the perspective of parsing and semantics generation.

For the above language, we can define a DCG G_3:

```
:- op(1000, xfy, <-).

n(X^bird(X)) --> [bird].
n((tweety^Sco)^Sco) --> [tweety].
```

[3] A more complete description of this initial lambda-calculus representation can be found in[4]

```
vi(X^fly(X)) --> [fly];[flies].
vcop(X^bird(X)) --> [is,a,bird];[are,birds].


d((X^Res)^(X^Sco)^(Sco <- Res)) --> [every];[all].

np(Sem) --> d((X^Res)^Sem), n(X^Res).
np(Sem) --> n(Sem).

vp(Sem) --> vi(Sem).
vp(Sem) --> vcop(Sem).

s(Sem) --> np((X^Sco)^Sem), vp(X^Sco).

go(Sen, Sem) :- string(Sen), s(Sem, Sen, []).

string([every,bird,flies]).
string([tweety,is,a,penguin]).
```

Their semantics can be generated by issuing the following two queries:

```
:- go([every,bird,flies],Semantics).
:- go([tweety,is,a,bird],Semantics).
```

Then, the meaning produced by the above grammar corresponds to the following first-order theory T_2:

$$(\forall x)(bird(s) \supset fly(s))$$
$$bird(tweety)$$

We can conclude that Tweety flies since $T_2 \models fly(tweety)$. In fact, the above theory T_2 can be further transformed into an extended logic program (a definite logic program in this case):

$$\begin{array}{rcl} fly(X) & \leftarrow & bird(X). \\ bird(tweety) & \leftarrow & . \end{array}$$

from which queries such as *what flies?*, *does tweety fly?* can derive and get the intended responses.

As discussed in the literature on non-monotonic reasoning, first-order theory is not adequate for meaning representation of the above example if we slightly modify it into the following text E_3:

Every bird flies. All penguins are birds. No penguin flies. Tweety is a penguin.

For the moment we don't consider conjunction words such as *although*, *but*, *and*, etc. in texts. If we follow the traditional method and use first-order theory to represent the meaning of the above sentence, we would have a definite clause grammar G_4:

```
:- op(400,fy,~).
:- op(700, xfy, <-).

n(X^bird(X)) --> [bird];[birds].
n(X^penguin(X)) --> [penguin];[penguins].
name(tweety) --> [tweety].

vi(X^fly(X)) --> [fly];[flies].
vcop(X^bird(X)) --> [is,a,bird];[are,birds].
```

```
vcop(X^penguin(X)) --> [is,a,penguin];[are,penguins].

d((X^Res)^(X^Sco)^(Sco <- Res)) --> [every];[all].
d((X^Res)^(X^Sco)^(~Sco <-  Res)) --> [no].

np(Sem) --> d((X^Res)^Sem), n(X^Res).
np((X^Sco)^Sco) --> name(X).

vp(Sem) --> vi(Sem).
vp(Sem) --> vcop(Sem).
vp(X^Pred) --> vt(Y^X^Sco), np((Y^Sco)^Pred).

s(Sem) --> np((X^Sco)^Sem), vp(X^Sco).

go(Sent, Sem) :- string(Sent), s(Sem, Sent, []).

string([every,bird,flies]).
string([tweety,is,a,penguin]).
string([all,penguins,are,birds]).
string([no,penguins,fly]).
```

After parsing the above text against G_4 we will have the following theory T_3:

$$\begin{aligned}
&fly(X) \leftarrow bird(X)\\
&bird(X) \leftarrow penguin(X)\\
&\neg fly(X) \leftarrow penguin(X)\\
&penguin(tweety)
\end{aligned}$$

It is easy to see that we will have both $T_3 \models fly(tweety)$ and $T_3 \models \neg fly(tweety)$. This implies that the theory T_3 is not consistent. But what's wrong with it? The English text above seems to make perfect sense. The reason is simply as follows: When we speak, write, or listen to a natural language text, we often take for granted some common-sense assumptions. When we say that every bird flies, we actually mean that every bird normally flies. If we understand it in this way, the intended meaning of the text E_4 should be something like this T_3':

$$\begin{aligned}
&fly(X) \leftarrow bird(X), not\ ab(X)\\
&bird(X) \leftarrow penguin(X)\\
&\neg fly(X) \leftarrow penguin(X)\\
&penguin(tweety)
\end{aligned}$$

The first formula of T_3' reads: for every bird, if it is not abnormal, it flies. Now we can prove that $T_3' \models \neg fly(tweety)$, but can't prove that $T_3' \models fly(tweety)$. Therefore, no contradiction arises in T_3'. But this is not the end of the story. Suppose that we go back to text E_2, and represent the meaning of E_2 by the following theory T_2':

$$\begin{aligned}
&fly(X) \leftarrow bird(X), not\ ab(X)\\
&bird(tweety)
\end{aligned}$$

We can no longer $T_2' \models fly(tweety)$. However, in the text E_2 we do intuitively mean that the bird *tweety* flies.

The above discussion means that after $ab/1$ predicate is introduced, we can solve the inconsistency problem in an English text (namely, E_3), but we fail to derive what we should be able to derive in another English text (namely, E_2). The problem is that our entailment $\models$ is the first-order consequence relation. If we understand T_2' as a circumscriptive theory where the extension of $ab(X)$ is minimized in its model, then we are still able to conclude that $tweety$ flies. That is, the meaning of E_2 should be represented by the following circumscriptive theory $Circ(T_2'; abnormal)$. Equivalently, we can represent the meaning of E_2 as the following logic program P_2:

$$\begin{array}{rcl} fly(X) & \leftarrow & bird(X), not\ ab(X) \\ bird(tweety) & \leftarrow & \end{array}$$

Then, the positive information in its WFSX model is

$$\{bird(tweety), fly(tweety)\}$$

And the meaning of E_3 can be represented by the following extended logic program P_3:

$$\begin{array}{rcl} fly(X) & \leftarrow & bird(X), not\ abnormal(X) \\ bird(X) & \leftarrow & penguin(X) \\ \neg fly(X) & \leftarrow & penguin(X) \\ penguin(tweety) & \leftarrow & \end{array}$$

If we do so and define its semantics as WFSX semantics, then we are still able to prove that there is a contradiction in it: Tweety flies and Tweety does not fly. This is because $ab(X)$ is not disabled. In order to solve this problem, we can introduce one more rule when we translate "all penguins are birds":

$$ab(X) \leftarrow penguin(X)$$

Then, positive information in its WFSX model is as follows:

$$\{penguin(tweety), \neg fly(tweety)\}$$

No inconsistency arises in it and the meaning agrees with our intuition.

In what follows we will further explore the above ideas, and develop a methodology to translate English texts into extended logic programs by using definite clause grammars. The objective is: given a definite clause grammar, we can parse an English text and generate its common-sense meaning at the same time. The extended logic program can then be executed in WFSX interpreter, and thus we can query about natural language texts.

5 Abnormality and its Diversity

In the previous section we have shown the need of the predicate $ab/1$ to resolve implicit exceptions. In order to automatically generate meanings of a natural language text through deductive parsing by using DCG, we have to solve the following problems:

- When a text is parsed, its meaning is built from those of its constituents. When a sentence like *all birds fly* is parsed, we actually add an additional assumption expressed by $ab/1$, and when the sentence like *no penguins fly* is parsed, we add a new formula $penguin(X) \rightarrow ab(X)$. It seems that one constituent may have more than what it explicitly delivers.
- Can we uniformly generate the meaning of a sentence like *all* A *does* B or A *does all* B by always adding an abnormality predicate? According to well-founded model semantics, if nothing is said about abnormality, then it is not abnormal by default. And thus, always adding an abnormality predicate won't lead to absurd conclusions.

- An object may be abnormal in more than one aspect. For example, a bird normally has two wings, and normally flies. And thus unary abnormality won't work, when it is translated into the following two programming rules:

$$\begin{aligned} fly(X) &\leftarrow bird(X), not\ abnormal(X) \\ has\_wings(X) &\leftarrow bird(X), not\ abnormal(X) \end{aligned}$$

  To handle this problem, it is sufficient to make use of a binary predicate, say $ab(asp, obj)$, to express an object obj is abnormal in aspect asp.

To summarize, as an example, we list the text, the DCG, and the extended logic program as follows:

English text: Every bird flies. Every bird has two wings. All penguins are birds. No penghins fly. Tweety is a penguin.

Extended logic program:

$$\begin{aligned} fly(X) &\leftarrow bird(X), not\ ab(fly, X) \\ has\_wings(X) &\leftarrow bird(X), not\ ab(has\_wings, X) \\ bird(X) &\leftarrow penguin(X), not\ ab(bird, X) \\ \neg fly(X) &\leftarrow penguin(X), not\ ab(\neg fly, X) \\ ab(fly, X) &\leftarrow penguin(X) \\ penguin(tweety) &\leftarrow \end{aligned}$$

DCG grammar:

```
:- op(950, fy, -). % Explicit negation
:- op(950, fy, not). % negation as failure
:- op(1110, xfy, '<-').  % Rule symbol
:- op(1110, yf, '<-'). % rule symbol
:- op(1110, fy, '<-'). % rule symbol

n(X^bird(X)) --> [bird];[birds].
n(X^penguin(X)) --> [penguin];[penguins].
name(tweety) --> [tweety].

vi(X^fly(X)) --> [fly];[flies].
vi(X^has_wings(X)) --> [has,two,wings].
vcop(X^bird(X)) --> [is,a,bird];[are,birds].
vcop(X^penguin(X)) --> [is,a,penguin];[are,penguins].

d((X^Res)^(X^Sco)^((Sco) <- (Res), not ab(Sco, X)))
                                         --> [every];[all].
d((X^Res)^(X^Sco)^(- (Sco) <-  (Res), not ab(-(Sco), X):
                              ab(Sco, X) <- (Res))) --> [no].
% We make use of : to mark the termination of an extended rule

np(Sem) --> d((X^Res)^Sem), n(X^Res).
np((X^Sco)^Sco) --> name(X).

vp(Sem) --> vi(Sem).
vp(Sem) --> vcop(Sem).

s(Sem) --> np((X^Sco)^Sem), vp(X^Sco).
```

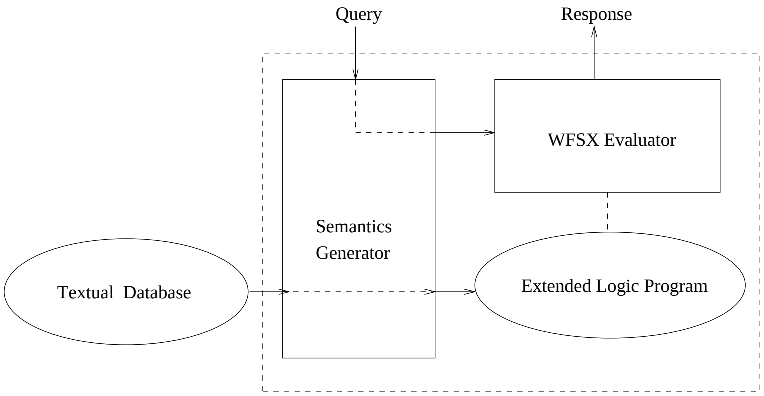

Figure 1: A Textual Database Query System

```
go(Sent, Sem) :- string(Sent), s(Sem, Sent, []).

string([every,bird,flies]).
string([every,bird,has,two,wings]).
string([all,penguins,are,birds]).
string([no,penguins,fly]).
string([tweety,is,a,penguin]).
```

6 Implementation

All examples in the previous sections have been tested with SICSTUS Prolog and an extended logic program WFSX meta-interpreter. The architecture of our system is illustrated in Fig. 1.

Technically a textual database is a collection of English texts and a lot of indexes on them. In this paper we simply regard a textual database as an English text such as a sentence, or a paragraph, or an article, or a story, or a WWW text page. The semantics generator is, basically, a logic grammar(e.g., a DCG program or Assumption Grammar [12]), whose execution results in an extended logic program taken as the semantics of the textual database. A query, still in the natural language form, is translated into an extended logic program goal, whose execution in the WFSX meta-interpreter will produce appropriate responses. Depending on the textual database, the DCG grammar may be very big. As meaning representation is a very complicated issue, in this paper we have restricted ourselves to the aspect of exceptions in texts. Other aspects, such as anaphora and coordination, can be treated as in [1].

7 Concluding Remarks

Representing and reasoning about natural language texts is a difficult and complicated task. In this paper we have addressed the semantic issue of exceptions in natural language texts. As natural language is often used by people with common sense, first-order logic is not adequate for meaning representation any longer. In this paper we have made use of extended logic programs for meaning representations, since extended logic programs allow for non-monotonic reasoning and defeasible conclusions and are executable. We have shown how to systematically generate semantics of exceptions through deductive parsing by using definite clause grammars. The method of this paper is especially applicable to incomplete textual databases or dynamic textual databases. but it is also generally applicable to the analysis of natural language descriptions in view of conceptual modelling and to natural language interfaces for database querying. Generally speaking, it also provides a method for conflict resolution when the conflict arises from exceptions. All examples have been tested with SICSTUS Prolog and a WFSX meta-interpreter [2, 3]. We should also point out that similar results could be obtained by using contradiction removal semantics of logic programs and preferences over *ab* predicates [3, 8]. Our future work will focus on a user-friendly interface to our query system, using Assumption Grammars.

Acknowledgement

This work was partially supported by NSERC of Canada under 31-611024 and JNICT of Portugal under PRAXIS 2/2.1/TIT/1593/95 and PRAXIS XXI/BPD/4165/94.

References

[1] H. Abramson and V. Dahl. *Logic Grammars.* Springer-Verlag, 1989.

[2] J. J. Alferes, C. V. Damásio, and L. M. Pereira. Slx - a top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *International Logic Programming Symposium.* MIT Press, 1994.

[3] J. J. Alferes and L.M. Pereira. *Reasoning with Logic Programming, LNAI 1111.* Springer, 1996.

[4] M. A. Covington. *Natural language processing for Prolog programmers.* Prentice-Hall, 1994.

[5] J. McCarthy. Applications of circumscription to formalizing common-sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[6] R. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[7] A. Nerode. *Proc. of the Third International Conference on Logic Programming and Non-Monotonic Reasoning, LNAI.* Springer, 1995.

[8] L. M. Pereira, J. N. Aparicio, and J. J. Alferes. Nonmonotonic reasoning with well founded semantics. In K. Furukawa, editor, *8th International Conference on Logic Programming.* MIT Press, 1991.

[9] L. M. Pereira and A. Nerode. *Logic Programming and Non-Monotonic Reasoning: Proc. of the Second International Workshop.* MIT Press, 1993.

[10] T. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and logic Programming*, pages 193–216. Morgan Kaufmann, 1987.

[11] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[12] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars. In *ILPS'95 Workshop on Visions for the Future of Logic Programming*, November 1995.