From: AAAI-88 Proceedings. Copyright ©1988, AAAI (www.aaai.org). All rights reserved.

Compiling Circumscriptive Theories into Logic Programs: Preliminary Report*

Michael Gelfond
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968

Vladimir Lifschitz
Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

We study the possibility of reducing some special cases of circumscription to logic programming. The description of a given circumscriptive theory T can be sometimes transformed into a logic program Π, so that, by running Π, we can determine whether a given ground literal is provable in T. The method is applicable, in particular, to some formalizations of tree-structured inheritance systems with exceptions.

1 Introduction

Circumscription was introduced by John McCarthy [1980; 1986] as a tool for formalizing the nonmonotonic aspects of commonsense knowledge and reasoning. A formula F follows from axioms A by circumscription if F is true in all models of A that are "minimal" in a certain sense. There may be several different minimality conditions that can be applied in conjunction with a given axiom set, and, accordingly, there may be several different "circumscription policies" (forms of circumscription) C that can be applied to given axioms. To select a circumscription policy, we should specify which of the predicates available in the language are *circumscribed* (minimized) and which of the remaining predicates are *varied* in the process of minimization; also, *priorities* can be assigned to the circumscribed predicates.

Given a circumscriptive theory (A, C) and a formula F, we may wish to know whether F is a theorem of (A, C), that is, whether F follows from the axioms A by the circumscription represented by C. There is no general algorithm for this problem, and several authors have proposed computational methods for some special cases that are important for applications to AI. Many of these methods [Bossu and Siegel, 1985; Gelfond and Przymusinska, 1986; Przymusinski, 1986; Ginsberg, 1988] are, in essence, extensions of the query evaluation procedures used in logic programming.

In this paper we explore another approach to the use of logic programming for the automation of circumscription: *compiling circumscriptive theories into logic programs.*[1] We may be able to transform the given circumscriptive theory (A, C) and the goal formula F into a logic program Π and a query W, so that the output produced by Π for the query W will show whether F is a theorem of (A, C). The rules of the program Π will be essentially the axioms A, sometimes modified to reflect the circumscription policy C. In the simplest case, W will coincide with the goal F, and the answer *yes* will be interpreted as the conclusion that F is a theorem. In general, W will be obtained from F by a simple syntactic transformation.

We have to make rather strong assumptions about the form of the given circumscriptive theory and about the goal formula. Nevertheless, the method is applicable to a number of interesting examples, including, notably, some formalizations of tree-structured (i.e., not multiple) inheritance systems with exceptions.

The idea of reducing special cases of circumscription to logic programming is suggested by the well-known fact that minimization plays a fundamental role in the semantics of logic programs. The semantics of Horn clause programming defined by van Emden and Kowalski [1976] uses minimization of the same sort as in the definition of circumscription. The semantics of stratified programs with negation [Apt *et al.*, 1988; Van Gelder, 1988] is closely related to the use of priorities [Lifschitz, 1988; Przymusinski, 1988a; Przymusinski, 1988b].

The main differences between circumscription and the declarative semantics of logic programming can be summarized as follows.

1. In logic programming, different ground terms are assumed to represent different elements of the universe. There is no corresponding assumption in the definition of circumscription.
2. In logic programming, every predicate is minimized. In the definition of circumscription, some predicates are minimized, and others are not.
3. In logic programming, each given clause should be written as a "rule", with one of the atoms designated as the "head", and the rest included in the "body". Deciding whether a given predicate should be placed in the head or, negated, in the body, significantly affects the meaning of the program, because in the latter case the minimization of that predicate will be given a higher priority. The definition of circumscription, on the contrary, is invariant with respect to replacing axioms by logically equivalent formulas; the assignment of priorities is explicitly described by the circumscription policy.

In view of these differences, it is usually impossible to simply view the axioms of a circumscriptive theory as the rules of the corresponding logic program, and a compilation process is required.

In Section 2, we review some terminology and notation related to circumscription and logic programs. In Section

*This research was partially supported by DARPA under Contract N0039-82-C-0250.

[1] In [Gelfond, 1987] a similar method is applied to answering queries in autoepistemic theories.

3, a series of examples is given in which circumscriptive theories are translated into logic programs. In Section 4 we state a theorem that demonstrates the correctness of the method used in these examples. The full paper will contain the proof of the theorem and some extensions.

2 Terminology and Notation

We start with a fixed first-order language with a finite number of object, function and predicate constants. In this preliminary report we assume that there are no functions in the language, so that its only ground terms are object constants $C_1, C_2, \ldots$. In this case we call the formulas $C_i \neq C_j$ $(i < j)$ the *uniqueness of names axioms* for this language.

An *atom* is an atomic formula not containing equality. A *literal* is an atom (*positive* literal) or a negated atom (*negative* literal). A *clause* is a disjunction of literals. A clause is *negative* if each of its literals is negative, and *definite* if it has exactly one positive literal. A *rule* is a formula of the form

$$L_1 \wedge \ldots \wedge L_m \supset A,$$

where $L_1, \ldots, L_m$ $(m \geq 0)$ are literals (they form the *body* of the rule), and A is an atom (the *head*). A clause that has k positive literals can be written as a rule in k essentially different ways, because any of the k positive literals can be placed in the head. In particular, a negative clause cannot be written as a rule, and a definite clause corresponds to a single rule.

A *program* is a finite set of rules. We identify a program with the conjunction of its rules. The *definition* of a predicate P in a program Π is the subset of Π consisting of all rules that contain P in the head. A *stratification*[2] of Π is a partition of its predicates into disjoint parts

$$P^1; \ldots; P^k$$

such that, for every predicate P from P^i $(1 \leq i \leq k)$, (a) all predicates that occur in the definition of P belong to $P^1, \ldots, P^i$, and (b) all predicates that occur in the definition of P under $\neg$ belong to $P^1, \ldots, P^{i-1}$. It is convenient to allow some of the parts P^i to be empty. A program is *stratified* if it has a stratification.

If A is a sentence, and P, Z are disjoint lists of predicates, then Circum$(A; P; Z)$ stands for the result of circumscribing the predicates in P relative to A, with the predicates in Z allowed to vary [Lifschitz, 1985]. If P is broken into parts $P^1, \ldots, P^k$, then the circumscription assigning a higher priority to the members of P^i than to the members of P^j for $i < j$ is denoted by

$$\text{Circum}(A; P^1 > \ldots > P^k; Z).$$

The last argument Z will be omitted if it is empty. Notice that we use semicolons to separate the arguments of Circum from each other, whereas commas are used to separate predicates inside each of the lists $P^1, \ldots, P^k, Z$.

If Π is a stratified program without functions then, according to [Przymusinski, 1988b], its semantics can be characterized as follows: a sentence F in the language of Π is true relative to Π if, in the presence of the uniqueness of names axioms, it follows from the circumscription

$$\text{Circum}(\tilde{\forall}\Pi; P^1 > \ldots > P^k),$$

where $\tilde{\forall}$ denotes universal closure, and $P^1; \ldots; P^k$ is a stratification of Π. Denote this circumscription by Π'.

Given a stratified program Π and a ground atom W, a logic programming interpreter is supposed to answer *yes* if W is true relative to Π, and *no* if $\neg W$ is true. Accordingly, we define:

$$Ans(\Pi, W) = \begin{cases} yes, & \text{if } U \wedge \Pi' \models W; \\ no, & \text{if } U \wedge \Pi' \models \neg W; \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

where U is the conjunction of the uniqueness of names axioms. The third case corresponds to the situation when neither W nor $\neg W$ follows from the circumscription. According to [Przymusinski, 1988b], this is only possible for floundered queries.

This semantics differs from the iterated fixed point semantics [Apt *et al.*, 1988; Van Gelder, 1988] in that the latter takes into account Herbrand models only.

If W is a ground atom whose predicate does not belong to the language of Π then we set $Ans(\Pi, W) = no$.

3 Examples

Example 1. Consider the circumscriptive theory with the axioms:

$$John \neq Jack,\ John \neq Jim,\ Jack \neq Jim, \quad (1)$$

$$father(John, Jack), \quad (2)$$

$$father(Jack, Jim), \quad (3)$$

$$father(x, y) \wedge father(y, z) \supset grandfather(x, z), \quad (4)$$

with both predicates *father* and *grandfather* minimized. How can we use logic programming to determine whether a given ground literal in the language of this theory is a theorem? Consider the logic program Π whose rules are (2), (3) and (4). If the goal formula is a ground atom W then W follows from axioms (1)—(4) by circumscription iff $Ans(\Pi, W) = yes$. If the goal formula is a negated ground atom, then let W be the goal formula with the negation sign removed; $\neg W$ follows from the axioms by circumscription iff $Ans(\Pi, W) = no$.

The translation process used in Example 1 is extremely simple: Π is obtained from the axiom set A by dropping some axioms, and W is obtained from the goal formula by dropping the negation sign, if there was one. The main reason why translating was so easy is that the circumscription policy in this example is the standard circumscription policy of Horn clause logic programming — minimizing all predicates.

Remark 1. It is essential that the uniqueness of names axioms (1) were initially included in the axiom set. Without them, it would be impossible to prove any negated ground atom, and such formulas as $father(John, John)$ would be undecidable.[3] At the same time, it is essential that these

[2] This is essentially the definition from [Apt *et al.*, 1988], except that we stratify *predicates*, rather than *rules*.

[3] To see why, consider a model of axioms (2)—(4) in which the universe is a singleton. The extents of all predicates in this model are minimal.

axioms were deleted in the process of compilation: Syntactically, they are not rules and cannot be included in a program.

Remark 2. If axiom (4) were written as a clause

$$\neg father(x,y) \vee \neg father(y,z) \vee grandfather(x,z), \quad (4')$$

then an additional step would be required: replacing this clause by the corresponding rule (4). Notice that clause (4′) is definite, so that it can be written as a rule in only one way.

In applications to formalizing commonsense reasoning, circumscription is often used to minimize "abnormality" [McCarthy, 1986]. In such cases, the language contains one or more abnormality predicates ab, $ab1$, $ab2$, ... These predicates express that their arguments are exceptional relative to some "default principles".

Example 2. The axioms are:

$$Tweety \neq Opus,\ Tweety \neq Joe,\ Opus \neq Joe, \quad (5)$$

$$bird(x) \wedge \neg ab(x) \supset flies(x), \quad (6)$$

$$bird(Tweety), \quad (7)$$

$$bird(Opus), \quad (8)$$

$$ab(Opus). \quad (9)$$

Axiom (6) expresses that normally birds can fly. The predicates ab and $bird$ are minimized; $flies$ is varied.[4]

We compile the given axiom set into a logic program Π in the same way as above, i.e., simply delete the uniqueness of names axioms (5). The answer given by a logic programming system to a query $P(c)$, where P is one of the predicates ab, $bird$ and $flies$, and c is one of the constants $Tweety$, $Opus$, and Joe, is interpreted as follows:

1. If $Ans(\Pi, P(c)) = yes$ then the given circumscription implies $P(c)$. In this way we conclude that the circumscription implies

   $$ab(Opus),\ bird(Tweety),\ bird(Opus),\ flies(Tweety).$$

2. If $Ans(\Pi, P(c)) = no$ and P is one of the circumscribed predicates ab and $bird$, then the circumscription implies $\neg P(c)$. In this way we get the conclusions

   $$\neg ab(Joe),\ \neg ab(Tweety),\ \neg bird(Joe).$$

3. If $Ans(\Pi, P(c)) = no$ and P is the varied predicate $flies$, then $P(c)$ is undecidable: The circumscription implies neither $P(c)$ nor $\neg P(c)$. We conclude that the formulas $flies(Opus)$ and $flies(Joe)$ can be neither proved nor refuted on the basis of the given axioms even using circumscription.

Remark 3. The program constructed in Example 2 is stratified. For instance, we can place $bird$ and ab in P^1, and $flies$ in P^2.

Remark 4. If axiom (6) were written as a clause

$$\neg bird(x) \vee ab(x) \vee flies(x),$$

then we would have a choice between two ways of writing it as a rule: (6) and

$$bird(x) \wedge \neg flies(x) \supset ab(x). \quad (6')$$

The second possibility would lead to a stratified program also (place $bird$ and $flies$ in P^1 and ab in P^2). But that program would not be satisfactory for our purposes: It answers *no* to the query $flies(Tweety)$, even though this query follows from the axioms by circumscription. We will see in Section 4 that the main result justifying the correctness of our method distinguishes between (6) and (6′) by demanding that, in the absence of priorities, *the circumscribed predicates belong to the first stratum* P^1.

Example 3. Replace axioms (8) and (9) in the previous example by the axioms

$$penguin(Opus), \quad (10)$$

$$penguin(x) \supset bird(x), \quad (11)$$

$$penguin(x) \supset \neg flies(x), \quad (12)$$

$$\neg flies(Joe). \quad (13)$$

Thus the new axiom set is (5)—(7), (10)—(13). We minimize ab, $bird$ and $penguin$, and vary $flies$. The transformation used in Examples 1 and 2 (dropping the uniqueness of names axioms) is not sufficient in this case for producing a program, because some of the remaining axioms, (12) and (13), are not rules. In fact, (13) is a negative clause, and (12), written as a clause, is negative also, so that it is impossible to write either as a rule. Some additional work is needed.

The key observation is that the remaining formulas (6), (7), (10)—(13) will become a program if we replace all occurrences of $\neg flies$ by a new predicate,[5] $\overline{flies}$. The rules of this program are (6), (7), (10), (11),

$$penguin(x) \supset \overline{flies}(x) \quad (12')$$

and

$$\overline{flies}(Joe). \quad (13')$$

This program, however, is not satisfactory for our purpose, because it treats $flies$ and $\overline{flies}$ as unrelated predicates. The information that they represent each other's negation is lost here. This can be fixed in the following way. Let us go back to the axiom set (6), (7), (10)—(13) and find all pairs of axioms that, written as clauses, can be resolved upon $flies$. There are 2 such pairs: (6), (12) and (6), (13). The resolvent of the first pair is the definite clause

$$\neg bird(x) \vee ab(x) \vee \neg penguin(x);$$

written as a rule, it becomes[6]

$$bird(x) \wedge penguin(x) \supset ab(x). \quad (14)$$

The resolvent of the second pair is the definite clause

$$\neg bird(Joe) \vee ab(Joe);$$

written as a rule, it becomes

$$bird(Joe) \supset ab(Joe). \quad (15)$$

[4] Another reasonable circumscription policy is to leave $bird$ fixed. Unfortunately, our method is not applicable to circumscriptions with fixed predicates.

[5] Similar transformations were used by several authors, beginning with Meltzer [1966].

[6] In view of axiom (11), the literal $bird(x)$ in this rule can be dropped. We will ignore "optimizations" of this kind.

We add the resolvents (14), (15) to the program that was obtained by introducing $\overline{flies}$. The result is the program:

$$bird(x) \wedge \neg ab(x) \supset flies(x), \quad (6)$$
$$bird(Tweety), \quad (7)$$
$$penguin(Opus), \quad (10)$$
$$penguin(x) \supset bird(x), \quad (11)$$
$$penguin(x) \supset \overline{flies}(x), \quad (12')$$
$$\overline{flies}(Joe), \quad (13')$$
$$bird(x) \wedge penguin(x) \supset ab(x), \quad (14)$$
$$bird(Joe) \supset ab(Joe). \quad (15)$$

If the goal literal F does not have the form $\neg flies(c)$ then the program is used for resolving F in the same way as before. We conclude that these facts follow from the axioms by circumscription:

$$\neg ab(Tweety),\ ab(Opus),\ \neg ab(Joe),$$
$$bird(Tweety),\ bird(Opus),\ \neg bird(Joe),$$
$$\neg penguin(Tweety),\ penguin(Opus),\ \neg penguin(Joe),$$
$$flies(Tweety).$$

About the formulas $flies(Opus)$ and $flies(Joe)$ we conclude that they do not follow from the axioms by circumscription. If W is $\neg flies(c)$ then we look at the answer to the query $\overline{flies}(c)$. If the answer is *yes* then W follows from the axioms by circumscription; if *no* then it does not. In our example, we get $\neg flies(Opus)$ and $\neg flies(Joe)$.

Finally, we will show that *prioritized circumscription* can be sometimes compiled into a logic program in essentially the same way.

Example 4. Let us make axiom (12) in Example 3 weaker and replace it by

$$penguin(x) \wedge \neg ab1(x) \supset \neg flies(x) \quad (12_0)$$

(normally, penguins cannot fly). The new abnormality predicate $ab1$ will be circumscribed at a higher priority than ab, in accordance with the familiar principle that more specific information in an inheritance system should be given a higher priority. Thus we give priority 1 to minimizing $ab1$, $bird$ and $penguin$, and priority 2 to minimizing ab; as before, $flies$ is varied. Replacing $\neg flies$ by $\overline{flies}$ gives

$$penguin(x) \wedge \neg ab1(x) \supset \overline{flies}(x). \quad (12_0')$$

The first of the two resolvents computed in the process of compilation will get an additional term:

$$\neg bird(x) \vee ab(x) \vee \neg penguin(x) \vee ab1(x). \quad (16)$$

This clause has 2 positive literals, $ab(x)$ and $ab1(x)$, so that we have to decide which of them should be placed in the head. We choose the form

$$bird(x) \wedge penguin(x) \wedge \neg ab1(x) \supset ab(x), \quad (14_0)$$

because the given circumscription policy assigns to $ab1$ a higher priority than to ab. Generally, we will require that the resulting program have a stratification *with the higher priority predicates placed in the lower strata.* This requirement determines how the assignment of priorities affects the result of compilation. The result of compilation is the program

$$bird(x) \wedge \neg ab(x) \supset flies(x), \quad (6)$$
$$bird(Tweety), \quad (7)$$
$$penguin(Opus), \quad (10)$$
$$penguin(x) \supset bird(x), \quad (11)$$
$$penguin(x) \wedge \neg ab1(x) \supset \overline{flies}(x), \quad (12_0')$$
$$\overline{flies}(Joe), \quad (13')$$
$$bird(x) \wedge penguin(x) \wedge \neg ab1(x) \supset ab(x), \quad (14_0)$$
$$bird(Joe) \supset ab(Joe). \quad (15)$$

Its answers are interpreted in the same way as in Example 3. If the predicate in the goal literal is ab, $bird$, $penguin$ or $flies$, then the result of computation is the same as before. To each query of the form $ab1(c)$ the program answers *no*, which shows that all these formulas can be refuted in the given circumscriptive theory.

4 Main Theorem

Let A be (the conjunction of) a set of clauses without function symbols. These clauses, along with the uniqueness of names axioms, will constitute the axiom set of the circumscriptive theory that we want to compile into a logic program. The circumscription policy of the theory will be determined by k disjoint lists of predicates $P^1, \ldots, P^k$ ($k \geq 1$) occurring in A. These predicates will be minimized: Those included in P^1 with the highest priority, those in P^k with the lowest. Let Z be the list of predicates $Z_1, \ldots, Z_l$ that occur in A but are not included in $P^1, \ldots, P^k$. These predicates will be allowed to vary. Symbolically, the circumscription under consideration is

$$\text{Circum}(\tilde{\forall} A \wedge U; P^1 > \ldots > P^k; Z),$$

where U is the conjunction of the uniqueness of names axioms. We will denote this formula by $Circum$.

We assume that every clause in A contains *at most one literal whose predicate symbol belongs to* Z.

We have seen that the process of compilation may include the replacement of some negated predicates by new predicate symbols, and also deriving new axioms by resolution. To describe these processes in the general form, assume that for each i ($1 \leq i \leq l$) a new predicate $\overline{Z}_i$ is selected, of the same arity as Z_i. The list of new predicates $\overline{Z}_1, \ldots, \overline{Z}_l$ will be denoted by $\overline{Z}$. By $Replace(A)$ we denote the result of replacing each $\neg Z_i$ in A by $\overline{Z}_i$. Let $Resolve(A)$ be (the conjunction of) the set of clauses that can be obtained by resolving a pair of clauses from A upon an atom whose predicate symbol belongs to Z. Since every clause in A contains at most one literal whose predicate belongs to Z, the formula $Resolve(A)$ does not contain predicates from Z.

Theorem 1. *Let* Π *be a program obtained from* $Replace(A) \cup Resolve(A)$ *by writing each clause as a rule, so that the partition*

$$P^1;\ \ldots;\ P^k;\ Z, \overline{Z} \quad (17)$$

is a stratification of Π. *Then, for any ground atom* W *whose predicate symbol occurs in* A,

1. $Circum \models W$ *iff* $Ans(\Pi, W) = yes$;
2. *If the predicate symbol of* W *belongs to* $P^1, \ldots, P^k$, *then*

$$Circum \models \neg W \quad iff \quad Ans(\Pi, W) = no;$$

3. *If the predicate symbol of* W *belongs to* Z, *then*

$$Circum \models \neg W \quad iff \quad Ans(\Pi, Replace(\neg W)) = yes.$$

It is easy to see that the conclusions made in Examples 1—4 above can be justified on the basis of Theorem 1. In Example 2, the predicate $\overline{flies}$ does not belong to the language of the program obtained as the result of compilation; this fact justifies our conclusion that no literal of the form $\neg flies(c)$ is a theorem.

Remark 5. There is no guarantee, of course, that each clause in $Replace(A) \cup Resolve(A)$ can be written as a rule stratified by (17). But there is a simple algorithm that transforms a given clause X into a rule stratified by (17) or determines that this is impossible. If X is negative, then the task is impossible. Otherwise, find the last among the groups (17) that contain a predicate occurring in X positively. If X has only one positive literal whose predicate is in that group, then make this literal the head of the rule. Otherwise the task is impossible.

An important class of examples in which some clauses cannot be stratified by (17) is given by *multiple* inheritance systems.

Example 5. The system formalized in Example 4 will become a multiple inheritance system if we disregard the fact that penguins are a subclass of birds, and treat *penguin* and *bird* as two partially overlapping classes. Formally, we consider the circumscriptive theory with the axioms (5)—(7), (10), (12_0), (13), and with *the same priority* assigned to the minimized predicates *ab*, *ab*1. Now both positive predicates in clause (16), obtained by resolving (6) against (12_0), belong to P^1. Hence there is no way to write that clause as a rule stratified by (17).

Remark 6. If A consists of n clauses, then $Replace(A)$ consists of n clauses too, and the number of clauses in $Resolve(A)$ is at most $(n^2 - n)/2$. Hence the number of rules in Π is at most $(n^2 + n)/2$.

Acknowledgments

We are grateful to Krzysztof Apt, Matthew Ginsberg, John McCarthy, Halina Przymusinska and Teodor Przymusinski for useful discussions.

References

[Apt *et al.*, 1988] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, CA, 1988.

[Bossu and Siegel, 1985] Genevieve Bossu and Pierre Siegel. Saturation, nonmonotonic reasoning and the closed-world assumption. *Artificial Intelligence*, 25(1):13–63, 1985.

[van Emden and Kowalski, 1976] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language, *Journal ACM*, 23(4):733–742, 1976.

[Gelfond, 1987] Michael Gelfond. On stratified autoepistemic theories. In *Proceedings AAAI-87*, 1, pages 207–211. Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Gelfond and Przymusinska, 1986] Michael Gelfond and Halina Przymusinska. Negation as failure: Careful closure procedure. *Artificial Intelligence* 30(3):273–288, 1986.

[Ginsberg, 1988] Matthew Ginsberg. A circumscriptive theorem prover: preliminary report. In these *Proceedings*.

[Lifschitz, 1985] Vladimir Lifschitz. Computing circumscription. In *Proceedings IJCAI-85*, 1, pages 121–127. Morgan Kaufmann Publishers, Los Altos, CA, 1985.

[Lifschitz, 1988] Vladimir Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann Publishers, Los Altos, CA, 1988.

[McCarthy, 1980] John McCarthy. Circumscription — a form of non-monotonic reasoning. *Artificial Intelligence* 13(1,2):27–39, 1980.

[McCarthy, 1986] John McCarthy. Applications of circumscription to formalizing commonsense knowledge, *Artificial Intelligence* 28(1):89–118, 1986.

[Meltzer, 1966] Bernard Meltzer. Theorem proving for computers: some results on resolution and renaming. *Comput. Journal*, 8:341–343, 1966.

[Przymusinski, 1986] Teodor Przymusinski, Query answering in circumscriptive and closed-world theories. In *Proceedings AAAI-86*, 1, pages 186–190. Morgan Kaufmann Publishers, Los Altos, CA, 1986.

[Przymusinski, 1988a] Teodor Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Los Altos, CA, 1988.

[Przymusinski, 1988b] Teodor Przymusinski. *On the declarative and procedural semantics of logic programs.* Preprint, University of Texas at El Paso, 1988.

[Van Gelder, 1988] Allen Van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann Publishers, Los Altos, CA, 1988.