Interpreting Courteous Logic Programs

Diploma Thesis in Computer Science

submitted by

Marc Dörflinger

Bludenz, Austria

Student ID 97-918-759

Department of Informatics

University Zürich, Switzerland

Prof. Abraham Bernstein

Advisor: Dr. Norbert E. Fuchs

Date of submission: May 12, 2005

Abstract

Courteous logic programming is an extension of logic programming. In addition to negation as failure, it allows logic programs to use classical negation in rule heads and bodies. As a consequence, logical conflicts can arise. Courteous logic programming introduces prioritization among the rules to resolve such conflicts. Furthermore, it introduces mutual exclusion between predicates.

In this thesis, we briefly discuss the theory of courteous logic programming and provide Prolog implementations of a forward and a backward chaining interpreter for courteous logic programs.

Furthermore, we show, how negation as failure can be removed from courteous logic programs, and present a precompiler, which does the removing process.

Courteous logic programming ist eine Erweiterung der logischen Programmierung. Sie erlaubt, zusätzlich zu negation as failure, die Verwendung von klassischer Negation in Kopf und Körper einer Regel. Als Folge davon können logische Konflikte entstehen. Courteous logic programming ermöglicht es, Regeln zu priorisieren, um solche Konflikte aufzulösen. Des Weiteren erlaubt sie den gegenseitigen Ausschluss von Prädikaten.

Im Rahmen dieser Arbeit beschreiben wir kurz die Theorie von courteous logic programming. Anschliessend implementieren wir je einen vorwärts- und einen rückwärtsschliessenden Interpreter in Prolog, welcher courteous logic programs versteht.

Zudem zeigen wir, wie negation as failure aus courteous logic programs entfernt werden kann und entwickeln dafür einen Precompiler.

Contents

Chapter 1

Introduction

During the last years, many people tried to make Prolog more expressive. There were different approaches of which one came from Benjamin N. Grosof in 1999. In his paper "Courteous Logic Programs" [Grosof97] he does not only extend logic programs by "classical negation", he also adds a mechanism to resolve conflicts, which can arise when classical negation is integrated into logic programs. This extension is called "courteous", because it respects precedence. It has priority relationships between rules and, in case of a conflict, the program is "well-behaved" and will always produce a consistent, tractably computable and unique answer set. Furthermore, courteous logic introduces mutual exclusion among predicates.

Grosof moreover developed a compiler, which transforms a courteous logic program into an ordinary logic program, which can then be interpreted by any normal Prolog interpreter, such as Sicstus or SWI-Prolog [Grosof99]. It is, however, the aim of this diploma thesis to develop an interpreter for courteous logic programs, which can directly interpret them without compiling them into ordinary logic programs first. In fact, there will be two interpreters, one using forward chaining and another one applying backward chaining. These interpreters will be written in Prolog so they can be used with any Prolog interpreter. For the sake of completeness and in order to provide a good overview, this study will also contain an introduction to courteous logic programming itself in Chapter 2. Chapter 3 shows, how the new language elements introduced by courteous logic are represented in Prolog. Chapters 4 - 5 will describe the theory and the implementation of the forward chaining interpreter as well as the transformation of negation as failure into classical negation and the implementation of the precompiler. It is moreover necessary to discuss the theory behind backward chaining and the implementation of the backward chaining interpreter, which will be subject of chapter 6.

Courteous logic programming contains two types of negation: classical negation and negation as failure. Since this can be very confusing to the user, this paper also aims at developing a precompiler to transform programs containing negations as failure into programs, which do not include negation as failure any more. After that transformation, a program will only make use of classical negation and will therefore be easier to understand, especially for unexperienced users. The transformed program makes use of skeptical reasoning, which is a feature of courteous logic programming.

Chapter 2

Courteous Logic Programming

2.1 Introduction to Courteous Logic Programming

2.1.1 Definitions

A courteous logic program is an extended logic program with additional features to make it more expressive. Extended logic programs are logic programs that include explicit negative information by using classical negation [Baral94]. The additional features of courteous logic programs are labels (to identify rules), overrides (to specify prioritization) and mutexes (to specify exclusions).

Notation and Terminology: From now on, we refer to a logic program as "LP" and to a courteous logic program as "CLP". The rule syntax of a courteous logic program is very similar to that of an extended logic program [Grosof97].

Definition 1 (Labeled rule) *Each labeled rule has the form*

$$\langle lab \rangle \qquad l_0 \quad \leftarrow \quad l_1 \wedge \cdots \wedge l_m \wedge \sim l_{m+1} \wedge \cdots \wedge \sim l_n$$

with lab *as the label for the rule and* $n \geq m \geq 0$ *and each* l_i *is a literal, which can be positive or negative. The* $\sim$ *stands for negation as failure. The label is optional and can be omitted.*

Labels are treated as an 0-ary function symbol and more than one rule can have the same label. All ground instances of a rule also carry the same label, since it is preserved during instantiation. □

Negation as failure is often called "soft negation". $\sim A$ is *true* in answer set S if $A \notin S$. In other words, $\sim A$ is *true* if it cannot be proved that A is *true*. Remember that with classical negation (so called "hard negation") $\neg A$ is *true* if $\neg A \in S$.

Since the head of a rule can be negated, there is a chance that two or more rules get in conflict with each other. A conflict arises, if both their bodies are true and their head are A and $\neg A$; this is called an "implicit mutex" (more on mutexes later). When using extended logic programs, the result set is inconsistent in case of an implicit mutex. Courteous logic programing solves this problem by allowing to specify which rule has the higher priority.

Definition 2 (Prioritization Predicate) *The predicate* overrides / 2 *is used for specifying prioritization of rules.* `overrides(I,J)` *means that the rule with label* I *has (strictly) higher priority than the rule with label* J. □

If a conflict arises during the instantiation of the rules, the prioritization rules are consulted to resolve the conflict. The rule with the higher prioritization refutes the other one. Rules that have no label have lowest priority; they will be refuted by labeled rules in any case. Two conflicting rules with no label cannot be refuted. If it is not determinable, which rule is refuted, then a "mutual defeat" is called and the program behaves "skeptically", which means that none of the possible outcomes will be added to the answer set. Due to this, there is no inconsistent answer set and the program goes on.

Definition 3 (Mutual Exclusion) *There are two types of mutual exclusion (short:* mutex*): "implicit mutual exclusion" (we already described that above) and "explicit mutual exclusion".*

Explicit mutual exclusions themselves are again divided into two forms: "conditional" or "unconditional mutexes.

An unconditional mutex has this syntax:

$$\bot \leftarrow l_1 \wedge l_2$$

where l_i are literals.

If both l_1 and l_2 are inferable from the rules, then the mutex is "active". The mutex is specified for any possible instantiation of the logical variables. Mutex's are useful for specifying different possible (allowed) alternatives of the result.

The conditional mutex is more general. It has the syntactic form:

$$\bot \leftarrow l_1 \wedge l_2 \mid g_1 \wedge \cdots \wedge g_g$$

where l_i denotes a classical literal and each g_i is a literal, which may contain $\sim$. The $g_1 \wedge \cdots \wedge g_g$ part of the formula specifies the condition in which the mutex is "active". When it is true, l_1 and l_2 oppose each other and should thus not be inferable at the same time. □

2.1.2 Other CLP Features

Database Updates Programs do not always stay the same. In time they will get updated with new (more recent) information, i.e. new rules. This can be done easily when using courteous LPs. The new rules only have to be added with higher priority than the previous rules. Database updates require no negation-as-failure and the number of prioritization facts, which need to be added, can be reduced by sharing the same label among more than one rule. This allows programs to be extended by "modules" later. As a side effect, updates can overturn older information directly or at least imply that information has to be overturned [Grosof97].

Code $\mathcal{C}_1$

$$\begin{aligned} &\langle prev \rangle \quad && q \leftarrow \\ &\langle prev \rangle \quad && p \leftarrow q \end{aligned}$$

with answer set $\{p, q\}$ gets later updated by

$$\begin{aligned} \langle upd \rangle \quad & \neg p \leftarrow \\ & overrides(upd, prev) \leftarrow \end{aligned}$$

Now the answer set is $\{q, \neg p\}$.

Theorem 1 (Forcing) *Courteous LPs have a relatively simple way to force a conclusion q: simply include the fact $q \leftarrow$ with higher priority than any other rule within its locale. Since there are no rules with higher priority, this rule will refute all others, hence it wins [Grosof97].* □

Theorem 2 (Cumulativity) *If p is a conclusion of the program $\mathcal{C}$, adding the fact $p \leftarrow$ (with any priority) to $\mathcal{C}$ results in a new program $\mathcal{C}'$ that has the same answer set as $\mathcal{C}$ [Grosof97].* □

Closed World Assumption Many databases as well as logic programming use the "Closed World Assumption" (CWA). CWA means that any p, which cannot be proved *true*, is assumed to be $false$.

When using CLP, the CWA for a given predicate p can be represented by adding the rule $\neg p(x)$ - with lowest priority within its locale - to the pool of rules.

Another approach to deal with CWA is shown by [Baral94]:

$$\neg p(x) \leftarrow\sim p(x)$$

If it cannot be proved that p is true, it is consequently false.

Example $\mathcal{C}_2$ An airline database which uses CWA to represent nonexistent flights [Grosof97].

$$\begin{array}{rl} \langle scheduled \rangle & flight(miami, detroit, 10am, elysian\_Air) \leftarrow \\ \langle scheduled \rangle & flight(jfk, new\_Orleans, 4pm, fountain\_Air) \leftarrow \\ \langle scheduled \rangle & flight(dallas, seattle, 7pm, middle\_Air) \leftarrow \\ \langle cwa\_flight \rangle & \neg flight(S, F, T, A) \leftarrow \\ & overrides(scheduled, cwa\_flight) \leftarrow \end{array}$$

The answer set of $\mathcal{C}_2$ consists of three positive $flight(\ldots)$ conclusions and a bunch of negative ones which were concluded by the CWA (for example $\neg flight(new\_Orleans, dallas, 4pm, middle\_Air)$).

2.1.3 Examples

Example 1 (Nixon Diamond) *[Grosof97]*

Code $\mathcal{E}_1$

$$\begin{array}{rl} \langle qua \rangle & pacifist(X) \leftarrow quaker(X) \\ \langle rep \rangle & \neg pacifist(X) \leftarrow republican(X) \\ & quaker(nixon) \leftarrow \\ & republican(nixon) \leftarrow \end{array}$$

When viewing $\mathcal{E}_1$ as a extended program, the first two rules are in conflict and render the program inconsistent. Applying the courteous interpretation, the program behaves skeptically and consistently and no conclusion is drawn regarding Nixon's pacifism. The answer is $\{quaker(nixon), republican(nixon)\}$.

After extending the program $\mathcal{E}_1$ by a rule, which gives higher priority to being a Republican than a Quaker,

$$overrides(rep, qua) \leftarrow$$

the answer set now is conclusive about Nixon's pacifism

$$\{\neg pacifist(nixon), quaker(nixon), republican(nixon)\}.$$

Example 2 (Mollusks) *[Grosof97]*

Code $\mathcal{E}_2$

$$
\begin{array}{rl}
 & mollusk(X) \leftarrow cephalopod(X) \\
 & cephalopod(X) \leftarrow nautilus(X) \\
\langle mol \rangle & shellbearer(X) \leftarrow mollusk(X) \\
\langle cep \rangle & \neg shellbearer(X) \leftarrow cephalopod(X) \\
\langle nau \rangle & shellbearer(X) \leftarrow nautilus(X) \\
 & overrides(nau, cep) \leftarrow \\
 & overrides(cep, mol) \leftarrow \\
 & overrides(nau, mol) \leftarrow \\
 & \\
 & mollusk(molly) \leftarrow \\
 & cephalopod(sophie) \leftarrow \\
 & nautilus(natalie) \leftarrow
\end{array}
$$

This example demonstrates the inheritance of mollusks by using prioritization rules. Without the prioritization rules, there would be a conflict when describing a cephalopod or a nautilus. The answer set for this program is

$$\{mollusk(natalie), mollusk(sophie), cephalopod(natalie), shellbearer(molly),$$
$$\neg shellbearer(sophie), shellbearer(natalie), mollusk(molly), cephalopod(sophie),$$
$$nautilus(natalie)\}.$$

Example 3 (Personal E-Mail Agents) *[Grosof97]*

The following example is a rule-based E-Mail Client for a person named Karen. Mail from a retailer should be trashed; however mail from someone Karen is awaiting a delivery from should be marked as important. This example illustrates how database updates can be done.

Code $\mathcal{E}_3$

$$
\begin{array}{rl}
\langle jun \rangle & \neg important(Msg) \leftarrow from(Msg, X) \wedge retailer(X) \\
\langle del \rangle & important(Msg) \leftarrow from(Msg, X) \wedge awaitingDeliveryFrom(karen, X)
\end{array}
$$

Additionally, Karen knows some facts to help the program sort the mail.

$$awaitingDeliveryFrom(karen, parisCo) \leftarrow$$
$$retailer(faveCo) \leftarrow$$
$$retailer(babyCo) \leftarrow$$
$$retailer(parisCo) \leftarrow$$

As soon as mail arrives from someone Karen is awaiting a delivery from, there is a conflict; it can be resolved by adding a rule to prioritize those messages:

$$overrides(del, jun) \leftarrow$$

If she decides later that she also wants mails from FaveCo to be classified important, this can be done easily by adding another rule. This is an example for a *database update*.

$$\begin{array}{ll} \langle fav \rangle & important(Msg) \leftarrow from(Msg, faveCo) \\ & overrides(fav, jun) \leftarrow \end{array}$$

After this extension, the answer set looks as follows:

$$\{\neg important(211), important(110), important(116),$$
$$awaitingDeliveryFrom(karen, parisCo), retailer(faveCo), retailer(babyCo),$$
$$retailer(parisCo), from(110, parisCo), from(116, faveCo), from(211, babyCo)\}.$$

Example 4 (Mail Importance: Family) *[Grosof97]*

This example, too, involves rules for mails. Fred wants mail from close family members to be of high importance, except messages from aunt Daisy. A notification about a personal emergency is of highest importance. The rules would therefore look like program $\mathcal{E}_4$.

Code $\mathcal{E}_4$

$$\begin{array}{ll} \langle clo \rangle & important(Msg) \leftarrow from(Msg, X) \wedge closeFamily(X, fred) \\ \langle dai \rangle & \neg important(Msg) \leftarrow from(Msg, auntDaisy) \\ \langle eme \rangle & important(Msg) \leftarrow notificationOf(Msg, Es) \wedge personalEmergency(Es) \\ & overrides(dai, clo) \leftarrow \\ & overrides(eme, dai) \leftarrow \end{array}$$

$$overrides(eme, clo) \leftarrow$$
$$personalEmergency(S) \leftarrow severeIllnessOf(S, X) \wedge closeFamily(X, fred)$$
$$closeFamily(betty, fred) \leftarrow$$
$$closeFamily(auntDaisy, fred) \leftarrow$$

A message from Betty

$$from(item19, betty)$$

would hence be marked as important ($important(item19)$), but one from Aunt Daisy on the other hand

$$from(item20, auntDaisy)$$

will not be ($\neg important(item20)$). This message from Daisy, however,

$$from(item115, auntDaisy) \leftarrow$$
$$notificationOf(item115, sit79) \leftarrow$$
$$severeIllness(sit79, auntDaisy) \leftarrow$$

will be marked as important, since it informs about an emergency. The complete answer set of this final example will be:

$$\{important(item115), personalEmergency(sit79), important(item19),$$
$$\neg important(item20), closeFamily(betty, fred),$$
$$closeFamily(auntDaisy, fred), from(item19, betty), from(item20, auntDaisy),$$
$$from(item115, auntDaisy), notificationOf(item115, sit79),$$
$$severeIllness(sit79, auntDaisy)\}.$$

Example 5 (Tweety - penguin or not?)

Last but not least, we present a widely recognized example often used in literature about artificial intelligence:

There are penguins and birds; penguins are birds despite the fact that they are not taken as the archetype of birds, since they cannot fly. A penguin should never be able to fly (higher priority). If something walks like a penguin, it is a penguin, as long as it has flat feet - if that is false, it is no penguin (higher priority). A third classification are wounded birds. A bird can never be wounded and fly at the same time (mutual exclusion).

We have three specimen of birds: Tweety, who is a bird, but not a normal one: he walks like a penguin, but does not have flat feet. Then there is Sam, who is a penguin, and John, a wounded bird. All this information put into rules appears as follows:

Code $\mathcal{E}_5$

$$
\begin{array}{rl}
\langle fly \rangle & fly(X) \leftarrow bird(X) \\
\langle nfly \rangle & \neg fly(X) \leftarrow penguin(X) \\
\langle peng \rangle & penguin(X) \leftarrow walkslikepeng(X) \\
\langle npeng \rangle & \neg penguin(X) \leftarrow \neg flatfeet(X) \\
\langle bird \rangle & bird(X) \leftarrow penguin(X) \\
\langle bird \rangle & bird(X) \leftarrow wounded\_bird(X)
\end{array}
$$

$$
\begin{array}{l}
bird(tweety) \leftarrow \\
walkslikepeng(tweety) \leftarrow \\
\neg flatfeet(tweety) \leftarrow \\
penguin(sam) \leftarrow \\
wounded\_bird(john) \leftarrow
\end{array}
$$

$$
\begin{array}{l}
overrides(npeng, peng) \leftarrow \\
overrides(nfly, fly) \leftarrow \\
\bot \quad \leftarrow fly(B) \wedge wounded\_bird(B) \mid bird(B)
\end{array}
$$

The answer set would be

$$
\{fly(tweety), \neg fly(sam), \neg penguin(tweety), bird(sam), bird(tweety), \\
walkslikepeng(tweety), \neg flatfeet(tweety), penguin(sam)\}
$$

This means that Tweety can fly, but Sam cannot; there is, however, no information about John and whether he can fly or not. Because of the mutual exclusion between being a wounded bird and flying, none of this information can be derived from the rules.

2.2 Courteous Compiler

The courteous compiler is a transformer which turns courteous logic programs (CLP) into ordinary logic programs (OLP). It supports mutual exclusion, classical negation and prioritized conflict handling [Grosof99]. Grosof decided to compile to OLP, because OLP are a very attractive target. They are computationally tractable, yet they support non-monotonicity via the negation as failure feature. Also many programmers are familiar

with those programs. The already available OLP rule systems and inferencing engines respectively are very efficient; most of them support forward as well as backward inferencing, which means that all needed functionality is already available.

Transforming CLP into OLP

Step 1: Eliminate classical negation by introducing a new predicate n_p for each $\neg p$ and a mutex between n_p and p ($\bot \leftarrow n\_p \wedge p$).

Step 2: Search for opposing predicates and mutexes.

Step 3: For each predicate q generate an associated set of OLP rules.

Step 4: Union the results of Step 3 for the overall output.

Since the details of this transformation are not of actual relevancy for this work, they are omitted. [1]

The courteous compiler is available as part of the IBM common rules software suite [CommonRules] and is also used within SWEET. [SWEET]

[1] For further information consult [Grosof99].

Chapter 3

Representation of Courteous Logic Program Elements in Prolog

In order to design a new Prolog interpreter which is able to interpret courteous logic programs, the first task is to decide how to represent the features of courteous logic programs introduced in chapter 2.1.1.

3.1 Labeled Rules

We represent CLP rules in Prolog in a format close to the format proposed by [Grosof97]:

$$\langle lab \rangle \quad l_0 \quad \leftarrow \quad l_1 \wedge \cdots \wedge l_m \wedge \sim l_{m+1} \wedge \cdots \wedge \sim ln$$

Namely as

```
lab ::  head(L,M) <- [body1(L), body2(M),...].
```

A fact, that is a rule without a body, is written as

```
lab ::  fact(A,B) <- [true].
```

or

```
lab ::  fact(A,B) <- .
```

Rules without labels can also be written by just starting the rule with `::`, e.g.

```
::  head(X,Y) <- [p(X), r(Y)].
```

Naturally, this also works for facts

```
::  fact(Z) <- .
```

Such rules will be translated internally into

```
emptyLabel ::  head(X,Y) <- [p(X), r(Y)].
```

3.2 Negation as Failure

In classical Prolog `\+` is used to denote negation as failure. Since Grosof uses $\sim$ in CLP rules, we decided to adopt $\sim$ for negation as failure.

3.3 Classical Negation

In the literature, $\neg$ is normally used as the symbol for classical negation. Within Prolog, however, this symbol is not appropriate due to the fact that it is not on the keyboard. `\+` is already taken as the symbol for negation as failure in Prolog, so the new operator `\-` for classical negation is more suitable.

E.g. a rule with negated head is written as

```
negHeadRule :: \-r(X) <- [p(X), q(3)].
```

3.4 Overrides Predicate

For the prioritization predicate *overrides/2* we apply a standard ordinary logic predicate, e.g. `overrides(Label1, Label2).`

3.5 Mutex

A mutex needs at least two arguments (the predicates, that are mutually exclusive) and maybe a condition. So mutex can be best described by the ternary predicate *mutex/3*. The third argument is the list of conditions which have to be fulfilled, if the mutex is conditional; in case of an unconditional mutex this list stays empty.

Thus, an example of an unconditional mutex looks like

```
mutex(p1(X), p2(Y), []).
```

and an example of a conditional mutex

```
mutex(q1(X), q2(Y), [c1(M), c2(N), ...]).
```

The condition `[c1(M), c2(N), ...]` works like the body of a rule. It is evaluated and if found true, the mutex is “active”.

Chapter 4

A Forward Chaining Interpreter for Courteous Logic Programs

4.1 Theory

Forward chaining is a bottom-up approach to find solutions to a problem. The idea of forward chaining is quite simple: find all facts you currently know for sure; then try to gather as much new information as possible by combining known facts and rules from the program. Add the newly learned information to the already known facts. Since you now have more information available, try to gather even more new information and so on. If you try to prove a certain goal, repeat this process until your goal is reached or no new information can be found, which means that your goal cannot be proved.

Forward chaining is "data driven" and therefore sometimes referred to as data driven inferencing [WikipFwChn]. [1]

4.2 Design

The purpose of a forward chaining interpreter for courteous logic programs is to find all possible answers A for the program S. The result is called "answer set" ($A \mid\sim S$). To achieve this, the interpreter has to repeat the following steps until no new answers can be found:

1. Select all facts. This is the initial knowledge base.
2. Fire all rules that can be fired with the knowledge base. Save the results in the working memory.

[1] For more details on forward chaining see chapter 9.3 of [Russel03].

3. Append new items in the working memory to the current knowledge base. In case of collision between two items, resolve the conflict by checking for overrides. If an item must be deleted in order to resolve a conflict, make sure all knowledge derived from it is deleted as well.

4. Check for mutexes.

5. If the contents of the knowledge base has changed during this iteration, go back to step 2.

Step 2 of this algorithm, i. e. the selection of rules that can be fired, is inefficient in the way it is currently handled. All rules, whose bodies are provable, are selected and fired. This behavior is inefficient, since the same rules may get fired multiple times with the same result. There is, however, an easy way to optimize this step. Only rules, whose bodies contain facts that have been derived in the last iteration, should be fired. Rules that rely entirely on facts known before the last iteration would have been fired already, so it does not make any sense to fire them again. Using this rule will accelerate the inferencing process considerably, since normally only a small number of rules depend on newly derived facts.

To demonstrate this algorithm, we use the mollusks example introduced in chapter 2.1.3. Applying the same representation as used internally to save the data (see chapter 4.3.1) the initial knowledge base of known facts looks like

```
[(f1, mollusk(molly), [true]), (f2, cephalopod(sophie), [true]),
 (f3, nautilus(natalie), [true]) ]
```

After the first iteration the working memory is

```
[(m, mollusk(sophie), [cephalopod(sophie)]),
 (c, cephalopod(natalie), [nautilus(natalie)]),
 (mol, shellbearer(molly), [mollusk(molly)]),
 (cep, \-shellbearer(sophie), [cephalopod(sophie)]),
 (nau, shellbearer(natalie), [nautilus(natalie)]) ]
```

Since all items in the working memory are unknown, they are all added to the knowledge base, which is now

```
[(m, mollusk(sophie), [cephalopod(sophie)]),
 (c, cephalopod(natalie), [nautilus(natalie)]),
```

```
 (mol, shellbearer(molly), [mollusk(molly)]),
 (cep, \-shellbearer(sophie), [cephalopod(sophie)]),
 (nau, shellbearer(natalie), [nautilus(natalie)]),
 (f1, mollusk(molly), [true]), (f2, cephalopod(sophie), [true]),
 (f3, nautilus(natalie), [true]) ].
```

After the second iteration the working memory is

```
[(m, mollusk(natalie), [cephalopod(natalie)]),
 (m, mollusk(sophie), [cephalopod(sophie)]),
 (mol, shellbearer(sophie), [mollusk(sophie)]),
 (mol, shellbearer(molly), [mollusk(molly)]),
 (cep, \-shellbearer(natalie), [cephalopod(natalie)]),
 (cep, \-shellbearer(sophie), [cephalopod(sophie)]) ]
```

But only `(m, mollusk(natalie), [cephalopod(natalie)])` is appended to the knowledge base, because all other facts are overridden by higher prioritized ones. During the the third iteration no new information can be gathered and the algorithm therefore stops; the final knowledge base is

```
[(m, mollusk(natalie), [cephalopod(natalie)]),
 (m, mollusk(sophie), [cephalopod(sophie)]),
 (c, cephalopod(natalie), [nautilus(natalie)]),
 (mol, shellbearer(molly), [mollusk(molly)]),
 (cep, \-shellbearer(sophie), [cephalopod(sophie)]),
 (nau, shellbearer(natalie), [nautilus(natalie)]),
 (f1, mollusk(molly), [true]), (f2, cephalopod(sophie), [true]),
 (f3, nautilus(natalie), [true]) ].
```

4.3 Implementation

Generally courteous logic programming allows negation as failure in the body of rules, but since all rules containing negation as failure can be transformed into equivalent rules without explicit negation as failure (see chapter 5 for details on this transformation), we decided not to support negation as failure.

It might also be useful to know some abbreviations often used in the Prolog code: `KB` stands for knowledge base, `WM` for working memory and `RuleSet` is a list containing all rules. The rules in this list are in a different form compared to when they are

entered into the program. While a rule is normally written as `lab :: head(X) <- [bodyElements].`, it is internally saved as a tuple `(Label, Head, Body)`; `Body` is the list of body elements as written in the original rule. [2]

4.3.1 The Knowledge Base and the Working Memory

The knowledge base is the place where all information, gathered from the rules is saved. It is a list which contains tuples composed of the label, the instantiated head and the instantiated body of the rule.

For reason of optimization, we put all facts into the initial knowledge base. This way we can save one iteration of the interpreter, which would only collect facts. Moreover, we fire only rules, whose bodies contain elements found during the last iteration. If the initial working memory would have been empty in the beginning, this optimization would not work as planned.

For the simple program

```
r1 :: a <- .
r2 :: b <- [a].
r3 :: c <- [b].
```

the initial knowledge base is `[(r1, a, [true])]`. After the 2nd iteration of the interpreter, it changes to `[(r3, c, [b]), (r2, b, [a]), (r1, a, [true])]`.

The working memory is a list of tuples like the knowledge base. It contains the answers found during the last iteration before they are added to the knowledge base. It could also be called a temporary answer set.

Additionally, there is the working memory from the latest iteration. It is a list, which contains only the instantiated head of all rules that have been fired during the last iteration. It is used for deciding which rules shall be fired.

For the above example, after the first iteration, the working memory would be `[(r2, b, [a])]`.

[2] The complete code of the interpreter can be found in the appendix or on the enclosed CD; the file is called *fwchn.pl*.

4.3.2 The Main Loop

The main loop of the forward chaining interpreter is fairly simple.

```
answerset(_, KB, [], KB).
answerset(RuleSet, KB, LastWM, KB5) :-
    fire(RuleSet, KB, LastWM, WM),
    append_and_check_overrides(KB, WM, KB2, NewWM, DeleteList),
    cascade_delete_list(KB2, DeleteList, KB3),
    check_mutex(KB3, KB4)
    answerset(RuleSet, KB4, NewWM, KB5).
```

Provided with a set of rules (`RuleSet`), a knowledge base (`KB`) and the latest working memory (`LastWM`; this is needed for optimization which will be explained in chapter 4.3.3), the loop first fires all rules, which can be fired with the current knowledge. Then it saves newly found facts to the knowledge base. Due to conflicts, there might be elements that need to be deleted from the knowledge base again. Those elements are saved to the `DeleteList` and after collecting them all per iteration, they are deleted using a cascading delete. Next is the check for mutual exclusions; after that, the next iteration is started by calling `answerset` with the updated knowledge base and current working memory again. This is done until no new facts can be gathered (`LastWM` equals `[]`).

4.3.3 Firing the Rules

The firing of the rules happens by stepping through the `RuleSet`; thereby every rule is checked whether it is qualified to fire. If this is the case, fire it.

```
fire([], _, _, []).
fire([(Label,Head,Body)|Rest], KB, LastWM, WM3) :-
    check_rule_qualified(LastWM, Body),
    findall((Label,Head, Body), fire_rule(Label, Head, Body, KB), WM),
    fire(Rest, KB, LastWM, WM2),
    append(WM, WM2, WM3).
fire([_|Rest], KB, LastWM, WM) :-
    fire(Rest, KB, LastWM, WM).
```

During the explanation of the main loop it was stated that we need the working memory from the last iteration in order to optimize. In this place, the last working memory is brought in. As already mentioned in chapter 4.2, a rule should only fire if it

depends on an element that has been derived during the last iteration. We need the latest working memory to check if a rule is allowed to fire.

This optimization should speed up the inferencing process considerably, since normally only a small percentage of the rules are fired this way; duplicate results (firing the same rule with the same facts) are also avoided.

If the rule is allowed to fire, `findall` is used to discover all instances of the rule. To do this, the predicates in the body are unified with the facts in the knowledge base. This is done with all rules that qualify and all results are put together to a list (`WM3`). This list (which may contain items that have already been derived in another loop or even might conflict with each other) is then handed back to the main loop.

We can demonstrate the rule selection behavior with a simple example program

```
r1 :: a <- .
r2 :: b <- [a].
r3 :: c <- [b].
```

Before the main loop starts, the programs is sorted in rules and facts. The facts are used as the last found working memory. During the first iteration, only rule `r2` is fired, since `a` is member of the last found working memory. This will lead to the new result `(r2, b, [a])`. During the next iteration, only rule `r3` will be fired, since it is the only one that qualifies when looking at the last found working memory. The new found working memory will then be `(r3, c, [b])`.

4.3.4 The Selection of Valid Answers

The working memory handed back to the main loop after the firing can contain, as mentioned before, conflicting answers as well as answers that are already known. Now it is necessary to clean out the not needed results.

```
append_and_check_overrides(KB, [], KB, [], []).
append_and_check_overrides(KB, [(Label, Head, Body)|Rest],
                           NewKB, NewWM, DeleteList) :-
    append_and_check_overrides(KB, Rest, TmpKB, TmpWM, TmpDL),
    (
        member((_,Head, _), TmpKB)
        ->
```

```
        NewKB = TmpKB,
        NewWM = TmpWM,
        DeleteList = TmpDL
    ;
        negate(Head, NegHead),
        (
            member((NegLabel, NegHead, NegBody), TmpKB)
            ->
            resolve_conflict((Label, Head, Body),
                             (NegLabel, NegHead, NegBody),
                             TmpKB, TmpWM, TmpDL, NewKB, NewWM,
                             DeleteList)
        ;
            append([(Label, Head, Body)], TmpKB, NewKB),
            append([Head], TmpWM, NewWM),
            DeleteList = TmpDL
        )
    ).
```

The algorithm steps through each of the elements in the result list and first checks, if the result is an already known fact, which has been derived in an earlier iteration. Is the result already known, it is skipped and the algorithm moves to the next element. If the result is unknown, however, the algorithm searches the contents of the knowledge base for the negated version of the element. If not found, the element is added to the knowledge base; if it is found, there is a conflict which needs to be resolved. If the conflict resolving process decides that an element must be deleted, the element will not be deleted instantly, but added to the `DeleteList` and is removed only after the working memory is added to the knowledge base completely. If the deletion would have happened before the working memory is complete, the cascading delete could miss elements depending on the removed object, which are added to the knowledge base later.

4.3.5 Resolving Conflicts

A conflict arises, when the negated version of an element is added to the knowledge base; for example this case: `(r1,p(1),[true])` exists already in the knowledge base and we add `(r2,\-p(1),[true])`. To resolve this conflict, we must look for a prioritization predicate `overrides(X,Y)`, that includes labels `r1` and `r2`. Three situations can appear now:

1. *The new element has lower priority than the one existing in the knowledge base.*

   In this case: do nothing! The higher prioritized element is already in the knowledge base, ignore the other one.

```
resolve_conflict((Label, _, _), (CLabel, _, _),
                 KB, WM, DL, KB, WM, DL) :-
    overrides(CLabel, Label).
```

   This case would apply, if the example above included `overrides(r1, r2)`.

2. *The new element has higher priority than the one existing in the knowledge base.*

   In this case, the element already existing in the knowledge base `(CLabel, CHead, CBody)` is refuted and the new one `(Label, Head, Body)` is added. The one, which must be deleted, is added to the delete list (`NewDL`), whereas the new one is added to the working memory.

```
resolve_conflict((Label, Head, Body), (CLabel, CHead, CBody),
                 KB, WM, DL, NewKB, NewWM, NewDL) :-
    overrides(Label, CLabel),
    append([(CLabel, CHead, CBody)], DL, NewDL),
    append([(Label, Head, Body)], KB, NewKB),
    (
        member(CHead, WM)
        ->
        delete(WM, CHead, TWM),
        append([Head], TWM, NewWM)
    ;
        append([Head], WM, NewWM)
    ).
```

   That case would apply, if the example above included `overrides(r2,r1)`.

3. *No element has higher priority; no prioritization information is available.*

   If that is the case, no element can be refuted. A non-courteous logic program would now have reached an inconsistent state and terminate. Since we implement a courteous interpreter, we can use the courteous solution and return the "skeptical answer set", which means that none of the two elements is added to the answer set.

The element, which already exists in the knowledge base `(CLabel, CHead, CBody)`, must be deleted, whereas the new one is ignored. Finally, the existing element is added to the delete list (`NewDL`).

```
resolve_conflict((_, Head, _), (CLabel, CHead, CBody),
                 KB, WM, DL, KB, NewWM, NewDL) :-
    write('Skeptical answer set returned.'), nl,
    append([(CLabel, CHead, CBody)], DL, NewDL),
    (
        member(CHead,WM)
        ->
        delete(WM, CHead, NewWM)
    ;
        NewWM = WM
    ).
```

This case would apply, if no prioritization information had been provided for the example above.

As mentioned in chapter 2.1.1, rules with no label have lowest priority. To make this work, it is necessary to add this override predicate - `overrides(_, emptyLabel).`

When returning to the main loop, the elements listed in the deleted elements list are deleted. To make sure all dependencies are removed as well, a cascading delete is used, as described in chapter 4.4.1.

In order to ensure that conflicts involving rules with label `emptyLabel` are resolved correctly, the rules above must check if the label is `emptyLabel`. We decided to omit a detailed description of this, as it might confuse the reader. For further details, look at the final version of the interpreter, which can be found in file *fwchn.pl* in the appendix or on the CD enclosed.

4.3.6 Checking for Mutual Exclusions

The first approach to find mutual exclusions was to iterate through the knowledge base and check on each element, whether it is involved in a mutex. By doing so, however, the order of the mutexes was no longer guaranteed.

So we decided to check all mutex's after every iteration of the main loop. If a mutex including two elements from the knowledge base is found, its condition is checked. If the

condition is empty (`[]`), the mutex is immediately active; if the condition list is populated with elements, we check whether the condition is provable with the information currently saved in the knowledge base. If this is the case, the mutex is active and the two elements are deleted from the knowledge base. As seen before, a cascading delete is used to remove the elements from the knowledge base.

As an example, look at this program

```
r1 :: a <- .
r2 :: b <- .
r3 :: c <- [a].
r4 :: d <- [a].
mutex(b, c).
```

After two iterations the answer set is `{a, d}`, if the mutex was `mutex(a, c)`, the answer set would have been just `{b}`.

4.4 Problems and Solutions

4.4.1 Cascading Delete

When an element is removed from the knowledge base, it is possible that the removal invalidates other elements in the list. Since they depend on the removed element, they cannot be proved anymore, after it was deleted; those elements must therefore be removed as well. This is called a cascading delete.

An example should make this clear:

```
a :: p(1) <- .
b :: x(1) <- .
c :: q(X) <- [p(X)].
d :: r(X) <- [q(X)].
e :: y(X) <- [x(X)].
f :: \-p(X) <- [y(X)].
overrides(f, a).
```

After the third iteration and before adding the working memory, the knowledge base looks like `[(a, p(1), [true]), (b, x(1), [true]), (c, q(1), [p(1)]), (e, y(1), [x(1)])]` and the working memory contains `[(d, r(1), [q(1)]), (f, p(1),`

`[y(1)])]`. There is a conflict between `p(1)` and `p(1)`. Due to the override `p(1)` is refuted and has to be deleted. Moreover `q(1)` and `r(1)` were inferred from `p(1)`; since that does not exist anymore, `q(1)` and `r(1)` need to be removed as well.

The cascading delete algorithm:

1. Delete the element (`DelItem`) from the knowledge base.
2. Delete all elements depending on it from the knowledge base. Save the heads of those elements in a list (`Deleted`).
3. For each element in `Deleted`, do a cascading delete.

The Prolog code for a cascading delete is (remember that the knowledge base is a list of tuples `(Label, Head, Body)`):

```
cascade_delete(KB, DelItem, NewKB) :-
    delete(KB, DelItem, TKB),
    cascade_delete(TKB, DelItem, Deleted, TmpKB),
    cascade_delete_list(TmpKB, Deleted, NewKB).
cascade_delete([], _, [], []).
cascade_delete([(DLabel, DHead, DBody)|Rest], (Label, Head, Body),
                [(DLabel, DHead, DBody)|Deleted], NewKB) :-
    member(Head, DBody),
    cascade_delete(Rest, (Label, Head, Body), Deleted, NewKB).
cascade_delete([(KLabel, KHead, KBody)|Rest], DelItem, Deleted,
                [(KLabel, KHead, KBody)|NewKB]) :-
    cascade_delete(Rest, DelItem, Deleted, NewKB).
cascade_delete_list(KB, [], KB).
cascade_delete_list(TmpKB, [D|R], NewKB) :-
    cascade_delete(TmpKB, D, TmpKB2),
    cascade_delete_list(TmpKB2, R, NewKB).
```

4.4.2 Recursive Programs

When trying to interpret recursive programs, the interpreter can go into an endless loop. An example for a program, which is running until no more memory is available, is this:

```
:: natnum(0) <- .
:: natnum(s(X)) <- [natnum(X)].
```

The first solution, the interpreter finds, is `natnum(0)`, then `natnum(s(0))`, after that `natnum(s(s(0)))`, and so on. The program has an infinite answer set.

Since forward chaining tries to find every answer of the program, forward chaining will not terminate in this case. With backward chaining, it would be possible to search for one specific answer to this program.

4.5 Evaluation

The forward chaining algorithm described in this chapter produces exactly one answer set for a given courteous program. The answer set is consistent, since courteous logic is able to resolve conflicts by removing inconsistencies, as proved in chapter 5.2 of [Grosof97]. But forward chaining cannot be used with recursive programs (see chapter 4.4.2).

Now we calculate the computational complexity for the forward chaining interpreter. Let n be the size of the program $\mathcal{C}$ and m the size of $\mathcal{C}^{instantiated}$. How much larger is m than n? If $\mathcal{C}$ has no free variables, then $m = n$. If the Datalog condition (no function symbols with arity greater than 0) applies and v is the maximum number of variables per rule, then the size of the Herbrand base is $\mathcal{O}(n)$ and m is $\mathcal{O}(n^{v+1})$.

As long as m is finite, $\mathcal{C}$'s entire answer set can be calculated in time $\mathcal{O}(m^2)$. When $\mathcal{C}$ obeys the Datalog condition, it can be done in $\mathcal{O}(n^{2\cdot(v+1)})$ [Grosof97].

All these numbers apply only as long as there is no conflict. The resolution of a conflict is done in linear time, if no elements have to be deleted from the knowledge base. If it is necessary to do so, however, the complexity of the cascading delete is in the worst case $\mathcal{O}(n^2)$. So if conflicts appear during the forward chaining, the computational complexity is squared in the worst case and changes to $\mathcal{O}(m^4)$ and $\mathcal{O}(n^{4\cdot(v+1)})$ respectively.

As this worst case analysis shows, the interpreter is not really fast, but there are two things that should considered: On the one hand, there is the optimization for the inferencing of rules; only rules which depend on answers found during the last iteration are fired. Normally, only a small part of the rule set depends on newly found answers, so this will accelerate things a lot. On the other hand, we can assume that conflicts are rather rare, compared to the number of elements in the knowledge base. Conflict resolution will therefore not happen on every iteration and a cascading delete might be even rarer, since not all conflict resolutions require cascaded deleting of elements.

So when taking these two things into consideration, we assume that the runtime will not be as bad as the worst case predicts.

The current implementation of the cascading delete is probably not very efficient. It would be a lot faster to first generate a dependency tree and delete elements according to its information: ($\mathcal{O}(log \cdot n)$ to look up dependent rules and $\mathcal{O}(n)$ to delete them). Nevertheless the effort might not be worth the overall performance gain.

After the theoretical formulas, we would also like to provide some actual run times. All measurements were conducted on a computer with an AthlonXP 2600+ processor (running at 1833 MHz)and 1GB RAM, running Gentoo Linux[3] with kernel 2.6.10. SWI-Prolog 5.4.6[4] was used as the Prolog interpreter. Three programs were run: the mollusk example (*example2.pl*) and two randomly created examples (*ruleset4.pl* and *ruleset5.pl*). The code for these programs can be found in the appendix. The mollusk code has 8 rules and 9 elements in the answer set, *ruleset4.pl* has 16 rules and 186 elements and *ruleset5.pl* has 23 rules and 1323 elements in the answer set.

Since one run of the mollusks example takes less than 1 millisecond (which is the smallest time span measurable in Prolog), it seemed appropriate to run every program 50 times. In order to achieve a good average, those 50 iterations were repeated 10 times.

The results are as follows: 50 iterations of the mollusks example took 41 ms on average, searching the answerset for *ruleset4.pl* 50 times took 2655 ms and the average time to find the 1323 elements for the answerset of *ruleset5.pl* was 110260 ms. To calculate one element in the answer set, the interpreter took about 0.09 ms in case of *example2.pl*, 0.28 ms in case of *ruleset4.pl* and 1.66 ms in case of *ruleset5.pl*. Since the runtime depends on things such as the level of recursions inside the program, the number of facts in relation to number of rules, as well as the number of conflicts, we believe these numbers are at most a good estimate.

[3]Information on Gentoo Linux can be found at [Gentoo]

[4]Information on SWI Prolog can be found at [SWI]

Chapter 5

Transformation of Negation as Failure to Classical Negation

5.1 Two Approaches to Eliminate Negation as Failure

Recently, there has been an increased interest in logic programs without negation as failure. Negation as failure together with classical negation can be quite conceptually confusing, especially for people who are not very familiar with the concepts. As we stated before in chapter 4.3, there are ways to transform programs, which contain negation as failure, into semantically equivalent programs, which do not contain negation as failure. Two such transformations are discussed in this chapter. It is therefore useful to repeat the difference between negation as failure and classical negation:

Negation as failure (often called "soft negation") uses "missing information" to negate an element. $\sim p$ is true, if there is no evidence, that p is true. A more formal way to write this: if A is the answer set of the program, then $\sim p$ is *true*, if $p \notin A$.

Classical negation (often called "hard negation") deals with real negative information. $\neg p$ means that p is *false*. Let A again be the answer set of the program. $\neg p$ is true, if $\neg p \in A$.

When using classical negation, we need to know certainly that something is *false*, whereas with negation as failure it is sufficient to know, that something is *not true*. Negation as failure is based on the absence of positive information, while classical negation is based on facts.

A simple example illustrates the difference: The sentence "A school bus is allowed to cross the railway tracks, when there is no train approaching" can be written as a logical rule in two ways. Either as a rule containing negation as failure

```
:: cross <- [~train]
```

or as a rule using classical negation

```
:: cross <- [\-train]
```

Both rules look similar, but there is a rather great difference. When using the negation as failure based rule, the bus is not allow to cross the tracks, in case a train is approaching. When there is no information about the train, the bus may cross the tracks. If for example the drivers view is blocked, the bus would be allowed to cross the tracks, since there is no information about an approaching train, but we do not want this to happen.

It is safer to use the rule applying classical negation. This rule requires explicit information about the train. The fact `:: \-train <- .` will have to be included in the program, if the bus wants to cross the tracks. In other words, the driver must know, that no train is approaching before crossing the tracks.

5.1.1 Transformation Using Closed World Assumption

Closed world assumption [Reiter78] states, that we can assume p is *false* ($\neg p$), when there is no positive information about p contained in the program. "Closed world assumption" (CWA) for rule `p` can be included in a program by adding this rule with lowest priority ([Gelfond91] and [Baral94]):

```
cwa :: \-p <- [~p].
```

If the rule is included, $\neg p$ and $\sim p$ can be used interchangeably. This would allow us to write a precompiler, which substitutes negation as failure in all rules by classical negation and then inserts the closed world assumption rule for all predicates. But since the closed world assumption rule itself is based on negation as failure, negation as failure is still not eliminated completely. Consequently this approach does not meet our demands.

5.1.2 Transformation Using Courteous Programs

A different approach to eliminate negation as failure from logic programs was suggested by [Antoniou98]. The approach described is based on skeptical reasoning, which is implemented by courteous logic programming.

A logic program $\mathcal{C}$, which contains rules using negation as failure, can be transformed into a semantically equivalent courteous program $\mathcal{C}'$ without negation as failure by applying a set of auxiliary predicates and classical negation. Every rule

$$\langle lab \rangle \qquad l \quad \leftarrow \quad l_1 \wedge \cdots \wedge l_n \wedge \sim m_1 \wedge \cdots \wedge \sim m_k$$

is replaced by the rules

$$\begin{array}{lrcl} \langle lab \rangle & l & \leftarrow & l_1 \wedge \cdots \wedge l_n \wedge p_r \\ & p_r & \leftarrow & \\ & \neg p_r & \leftarrow & m_1 \\ & & \cdots & \\ & \neg p_r & \leftarrow & m_k \end{array}$$

p_r is a propositional atom and is not part of the original program $\mathcal{C}$. Because no prioritization among these predicates is provided, skeptical reasoning is used in case of a conflict, which appears if one of the m_i rules is true, and consequently l cannot be true. Thus the answer set of the program $\mathcal{C}'$ stays the same as that of $\mathcal{C}$, but negation as failure is no longer used within $\mathcal{C}'$.

In chapter 5.2 a precompiler for transforming courteous logic programs containing negation as failure into negation as failure free courteous logic programs will be presented.

5.2 Implementation

5.2.1 Algorithm

The above transformation can be done using this algorithm:

1. Take next rule *rule* from the rule set $\mathcal{C}$.
2. Check whether the body of *rule* contains negation as failure. If this is the case, move on to step 4.
3. Add *rule* to new rule set $\mathcal{C}'$ and return to step 1.
4. Remove all elements with negation as failure from the body and save them in the list *Fails*.
5. Generate the name for the new propositional predicate by adding "prop_" in front of *rule*'s head and append a number at the end (in case there are more than one rules with the same head in $\mathcal{C}$).

6. Find all free variables *Fails* and save them to list *FreeVars*.

7. Generate a new positive rule p_r with the name generated in step 5 and add *FreeVars* as arguments. Add p_r to the new rule set.

8. For each element E found in *Fails*, generate a new negative predicate $\neg p_r$ with the same head as p_r and for the body take E. Add $\neg p_r$ to the new rule set $\mathcal{C}'$.

9. If there is at least one rule left in $\mathcal{C}$, go back to step 1.

Next it is useful to discuss the implementation of important parts of the algorithm described above. The full implementation can be found in the appendix or on the enclosed CD; the file is called *delfail.pl*.

5.2.2 Generating the Propositional Predicate

For the propositional predicate we need two things, arguments and a new name. As arguments, we take the free variables from the elements in *Fails*. All those variables contained before in the original rule will be used in the new rules, so p_r needs those variables, too.

The new name of p_r is constructed by taking the name of the original rule and adding "prop_" to the front of it. Since there can be multiple rules with the same name, it is necessary to add some additional unique letters to it. Simply add numbers to the name, 0 for the first rule, 1 for the second, etc.

Implemented in Prolog, the name generation process looks like this:

```
generate_prop_pred_name(Head, PropList, NewPropList, PropName) :-
    Head =.. L,
    nth0(0, L, Name),
    string_concat(prop_, Name, PN),
    (
        member((PN, Numb), PropList)
        ->
        Number is Numb + 1,
        delete(PropList, (PN, Numb), TmpPL),
        append([(PN, Number)], TmpPL, NewPropList)
    ;
        \+member((PN, _), PropList)
        ->
```

```
        Number = 0,
        append([(PN, Number)], PropList, NewPropList)
    ),
    string_concat(PN, Number, PN2),
    string_to_atom(PN2, PropName).
```

`Head` is the head of the original rule which is needed to generate the name. `PropList` is a list of already used rule names, which indicates how often they were used. This is necessary to acquire the number to add to the name. `NewPropList` is the same as `PropList` except that it contains the newly generated name or if the name was already used before, the number of times it was used in increased by one. `PropName` is the name for the propositional predicate.

Next is the list of free variables used by the elements in *Fails*. We therefore step through the list and check whether the arguments are variables (this is done with the Prolog predicate `var(X)`).

After acquiring this list, we can generate the new predicate p_r by concatenating the generated name and the variable list to a new list and then use the `=..`[1] predicate on it. The new predicate is now added to the original rule instead of the elements containing negation as failure.

5.2.3 Generating the Auxiliary Rules

These rules consist of one positive fact p_r and multiple negative rules $\neg p_r$. Each negative rule has *one* element in the body, an element from *Fails*. If no element in *Fails* is true, p_r stays true and the original rule can be fired; however, if one element in *Fails* becomes true, then p_r and $\neg p_r$ refute each other, since no prioritization information is available, and the original rule cannot fire.

The generation of these auxiliary rules implemented in Prolog:

```
add_aux_rules([], PropName, [(aux, PropName2, [true])]) :-
    copy_term(PropName, PropName2).
add_aux_rules([Fail|Rest], PropName, [PropTerm|AuxRules]) :-
    copy_term((aux,\-PropName, [Fail]), PropTerm),
    add_aux_rules(Rest, PropName, AuxRules).
```

[1] `..=` generates a new predicate from a list, by taking the first element of list as the predicate name and the remaining elements as arguments.

The first argument to the predicate is the *Fails* list, the second is the propositional predicate and the third is the list of generated rules. A copy of all terms is used to build the rules. In order to make sure, that the variables in the rules are not linked to each other when applied multiple times; otherwise, unwanted side effects may appear.

5.2.4 Example

Take this simple program

```
q1 :: q(X) <- [r(X), ~p(X)].
q2 :: q(X) <- [\-s(X), t(Y), ~v(X,Y)].
```

After letting it run through the preprocessor, it looks like

```
q1 :: q(A) <- [r(A), prop_p0(A)].
aux :: prop_q0(B) <- .
aux :: /-prop_q0(C) <- [p(C)].
q2 :: q(D) <- [\-s(D), t(E), prop_q1(D,E)].
aux :: prop_q1(F,G) <- .
aux :: \-prop_q1(H,I) <- [v(H,I)].
```

Also, the variables in the auxiliary rules are not the same as in the original rules anymore because of the `copy_term/2` predicate.

This example is "staged". Normally the input as well as the output have to be in knowledge base format. Since mostly one lets the program run through the interpreter, when using the preprocessor, we consider it more convenient to use the internal knowledge base format. The reformatting of the rules in this example has been done for better readability.

5.3 Evaluation

The program generated by the transformation is semantically equivalent to the original program. As long as it is interpreted with a courteous compiler, it will not produce any inconsistencies and there will only be one correct answer set.

New we are going to calculate the computational complexity. Let n be the number of rules in the program, p the maximum number of elements in the body and v the number of free variables used by the negation as failure elements. Finding and removing all negation as failure elements in the body can be done in time $\mathcal{O}(p)$. Detecting the free variables

takes at most $\mathcal{O}(p \cdot v)$, generating the auxiliary predicates lasts $\mathcal{O}(p \cdot v)$ time, if the worst comes to the worst.

Taken all this together and repeated for each rule in the program (n times), computational complexity of the algorithm is $\mathcal{O}(n \cdot p \cdot v)$ at the worst.

Chapter 6

A Backward Chaining Interpreter for Courteous Logic Programs

6.1 Theory

Backward chaining is the second method of reasoning, the top-down approach. While forward chaining (see chapter 4.1 for details) tries to deduct new information from existing information, backward chaining tries to prove that specific goals are true, by searching for rules and facts to support those goals [WikipBwChn].

When a goal is handed to the backward chaining interpreter, it searches for rules or facts, whose heads match the goal. If a fact is found, the goal is immediately proved true; on the other hand, if a rule is found, the interpreter has to prove, that all elements in the body of the rule are true. During this search and match process, variables get substituted by values. Once all goals and subgoals are proved, the interpreter finishes. The set of all provable goals should be the same as the answer set of the forward chaining algorithm for the same program. Due to its operation process, backward chaining is often called "goal driven". [1]

When we examine backward chaining in combination with courteous logic programs, things are getting tricky. The main question is, what happens when p and $\neg p$ appear in the same rule and in a dependent rule, respectively. To solve this problem, we have to survey the formulas behind the rules. To do this, we use the following rule set:

$$\begin{array}{llll} \langle r_1 \rangle & h_1 & \leftarrow & b_0 \wedge b_1 \wedge b_2 \wedge \ldots \\ \langle r_2 \rangle & b_0 & \leftarrow & p \wedge q_0 \wedge q_1 \wedge \ldots \end{array}$$

[1] For more details on backward chaining see [Russel03], chapter 9.4.

$$\langle r_3 \rangle \quad b_1 \quad \leftarrow \quad \neg p \wedge s_0 \wedge s_1 \wedge \ldots$$

When we now try to prove h_1 through backward chaining, we have to solve this formula

$$h_1 \quad \leftarrow \quad p \wedge \neg p \wedge q_0 \wedge q_1 \wedge \ldots \wedge s_0 \wedge s_1 \wedge \ldots \wedge b_2 \wedge \ldots$$

which can also be written as

$$h_1 \vee \neg(p \wedge \neg p \wedge q_0 \wedge q_1 \wedge \ldots \wedge s_0 \wedge s_1 \wedge \ldots \wedge b_2 \wedge \ldots)$$

which is equivalent to

$$h_1 \vee \neg p \vee p \vee \neg q_0 \vee \neg q_1 \vee \ldots \vee \neg s_0 \vee \neg s_1 \vee \ldots \vee \neg b_2 \vee \ldots$$

The final formula contains the sub formula $p \vee \neg p$, which is always true. With this sub formula it would therefore be possible to deduce anything from the formula, even false results. In any logic program, this would lead to an inconsistent state. Consequently p and $\neg p$ cannot be allowed to appear simultaneously. Hence, the backward chaining interpreter has to terminate as soon as the negated version of a term, which was already encountered, is found.

This, however, does not cover all cases. It is possible, that p and $\neg p$ are both true, but are not both used at the same time, since backward chaining needs only a part of the whole program to prove a fact. In order to get the same answers with backward and forward chaining, we have to check that p and $\neg p$ are not deductible simultaneously. So every time the interpreter finds a new term, it has to examine whether the opposite term is deductible in the overall program and if so, it must fail. This is the "skeptical behavior".

Next, we add the override mechanism to backward chaining; as discussed before (see chapter 2.1.1), overrides declare the prioritization among rules in case of a conflict. Again, if p and $\neg p$ are found in a rule at the same time, the rule can never be true if overrides are declared, since overrides make sure that only one of the two contesting elements is provable.

If a rule only depends on p or $\neg p$ and an override is declared, the rule can only be true, if the depending element has higher priority. In the following program, q can be proved through backward chaining, while r cannot.

$$\begin{array}{llll} \langle a \rangle & q & \leftarrow & p \\ \langle b \rangle & r & \leftarrow & \neg p \end{array}$$

$$\begin{aligned} \langle c \rangle \quad p \quad &\leftarrow \\ \langle d \rangle \quad \neg p \quad &\leftarrow \\ overrides(c, d) \quad &\leftarrow \end{aligned}$$

Finally, we will discuss how to integrate mutex features (see chapter 2.1.1 for details) into the interpreter. A mutex describes two elements that exclude each other and must not appear simultaneously. Additionally, there might be a condition, which must be true for the mutex to be active. As said before, backward chaining might only look at a small slice of the whole program to prove a query; so the second element of a mutex (given that the first is included within the context of that program slice) might not be looked at during the proof. Therefore, the mutex would appear to be not true, but it would again keep the answer set consistent. With the answer set of forward chaining, we must check, whether it would be possible to fulfill the second element of the mutex under the current circumstances. If so, the mutex is active and the interpreter has to stop. Additionally, if there is a condition to the mutex, we must examine first, whether that condition is provable.

6.2 Design

There are multiple ways to implement backward chaining, the two most known ones are similar to "depth-first search" (the goal list is a stack, elements are taken from the top, new elements are added to the top) and "breadth-first search" (the goal list is a queue, elements are taken from the beginning, new elements are added at the end). We decided to use depth-first search and implement the goal list as a "stack".

The initial *Stack* is the list of goals given by the user to prove. Then the following algorithm is used:

1. If *Stack* is empty, then end recursion call, else take the first element from *Stack* and save it in *Goal*
2. Search *RuleBase* for a rule r with a head matching *Goal*.
3. If r is a fact, go to step 4. If r is a rule, push the body of the rule onto *Stack*.
4. Make a recursive call to step 1.
5. After the recursive call, check whether the "negated version" of *Goal* is provable, if so the algorithm fails.

6. Check if a mutex for *Goal* is active, if so the algorithm fails.

   This is done after the recursive call, because before the recursive call there is the possibility, that variables have not been bound. This could for example lead to $p(X)$ and $\neg p(X)$ being opposing terms, even though after binding the variables maybe only $p(1)$ and $\neg p(2)$ would exist.

When trying to match the goal with a rule in step 2, there is a chance, that more than one choice is possible. This is called a "choice point". The interpreter takes the topmost answer first, but remembers the position, where a different choice could have been made. After completing the algorithm, it jumps back to that position and tries to find an alternative result by choosing a different matching rule.

As before, we show the behavior of the backward chaining algorithm by using the mollusks example introduced in chapter 2.1.3:

The query is `shellbearer(X)`, so the initial goal stack will be `[shellbearer(X)]`. During the recursions, the stack changes at first to `[mollusk(X)]`, then changes to `[cephalopod(X)]` and after that to `[nautilus(X)]`, which can then be matched with the fact `nautlius(natalie)`. While returning back to recursion level zero, all attempts to prove the opposite of the goal fail, until at level zero `mol::shellbearer(natalie)` is overridden by `cep::\-shellbearer(natalie)`, which has higher priority. So far, the result is not valid.

The interpreter has to step back to the last choice point, which is `[cephalopod(X)]`. The fact `cephalopod(sophie)` can be used as an alternative match, which would lead to the final result `shellbearer(sophie)`; this one is, however, conflicting with and overridden by `\-shellbearer(sophie)`. The next available choice point would be at recursion level one, where the fact `mollusk(molly)` can be matched. The upcoming result `shellbearer(molly)` is finally a valid one and presented as an answer of the query.

If the user decides to look for alternative results, the interpreter has to look for the next choice point. This is at level zero and leads to the alternative goal stack `[nautilus(X)]` at recursion level one. In the next recursion, a match is found and `shellbearer(natalie)` is presented as a next result. This time, the rule had higher priority and no opposer's were found, so the result is valid.

Since there are no more open choice points, no more alternative results are available. The results which we found are the same as we found when using forward chaining.

6.3 Implementation

As said before, we implement backward chaining using a stack, which contains the goal(s) that should be proved. The initial stack is only made up from the goals(s) included in the user query.

6.3.1 The Main Loop

The `prove` predicate is the main part of the backward chaining interpreter. It takes the first element of the stack and tries to match it with a fact or rule from the rule base. When a match is found, it checks whether a fact or rule was found. If it was a fact, the stack stays the same, if it was a rule, the body of the rule is pushed onto the stack. After that, a recursive call to `prove` is made. This is repeated until the stack is empty.

For optimization reasons, we keep track of all elements encountered so far. They are saved in the `Proved` and `NewProved` lists respectively. If we encounter an element which has already been proved, we can directly make the recursive call and look at the next element in the stack. If the prove of the element went wrong, the interpreter would not have come so far; this means we can be sure that we are in a consistent state. This is all, which is necessary for the backward chaining interpreter to prove a query, but since we interpret courteous logic programs, we must make sure that conflicts are resolved and mutexes are taken care of. This is done, when the recursive call is finished; before, there is the chance of unbound variables that might lead to a conflict. First we check for conflicts, after that we see if there are active mutexes. Those two jobs are described in the following chapters.

The implementation of the `prove` predicate is the following:

```
prove(GoalStack, RuleBase, Proved, NewProved) :-
    \+stack_is_empty(GoalStack),
    stack_pop(GoalStack, Goal, GoalStack2),
    member((_,Goal), Proved),
    prove(GoalStack2, RuleBase, Proved, NewProved).
prove(GoalStack, RuleBase, Proved, NewProved) :-
    \+stack_is_empty(GoalStack),
    stack_pop(GoalStack, Goal, GoalStack2),
    member((Label, Goal, Body), RuleBase),
    (
        Body == [true]
        ->
```

```
        GoalStack3 = GoalStack2
    ;
        Body \== [true]
        ->
        stack_push_many(Body, GoalStack2, GoalStack3)
    ),
    append(Proved, [(Label, Goal)], Proved2),
    prove(GoalStack3, RuleBase, Proved2, NewProved),
    check_for_opposer(Label, Goal, RuleBase, NewProved),
    check_for_mutex(Goal, RuleBase, NewProved).
prove(GoalStack, _, Proved, Proved) :-
    stack_is_empty(GoalStack).
```

6.3.2 Finding and Resolving Conflicts

Conflicts can arise in backward chaining either during the matching of rules, when p and $\neg p$ show up, but they can also appear "hidden". It might happen that p is used during the backward chaining; $\neg p$ would still be provable, but was not looked at during p's proof, since p's proof only made use of a slice of the program, which did not contain $\neg p$. This is a "hidden" conflict, which must be resolved as well.

For each element we prove, we must assure, that its opposite is not true by checking the `Proved` list. If that fails, we must try and prove the opposite. In case no opposing element is found, the interpreter can continue, otherwise we must check for overrides. If no override is specified, the interpreter behaves skeptically and stops. In case the override gives higher priority to the opposing element, the interpreter has to stop as well. If the opposing element has lower priority, the interpreter can continue.

The following code is the Prolog implementation for the detection of a conflict. The code, to resolve the conflict, can be found in the appendix or on the enclosed CD; the file is called *bwchn.pl*.

```
check_for_opposer(Label, Goal, RuleBase, Proved) :-
    negate(Goal, OppGoal),
    (
        member(OppGoal, Proved)
        ->
        fail
    ;
```

```
        \+member(OppGoal, Proved),
        findall(CLabel, prove_oppgoal(OppGoal, RuleBase, CLabel, Proved),
                CLabels)
        ->
        resolve_all_conflicts(Label, CLabels)
    ;
        true
    ).
prove_oppgoal(OppGoal, RuleBase, CLabel, Proved) :-
    member((CLabel,OppGoal, Body), RuleBase),
    (
        Body == [true]
        ->
        true
    ;
        Body \== [true]
        ->
        prove(Body, RuleBase, Proved, _)
    ).
```

6.3.3 Finding Mutexes

After making sure that no opposing predicate blocks the proving process, we must examine whether there are any mutexes that block the result.

To do this, we first have to find any mutexes involving the current element; then we must consider the condition. If the condition is not empty, we must check whether the condition is active, otherwise we can ignore the mutex. If the condition is active or there is no condition, we must examine whether the second element involved in the mutex is active. The procedure to do this is similar to that of checking for opposing elements. We first look whether we already encountered the element; if so, we already proved that it is true, if we did not encounter the element yet, we try to prove it. In case this proof does not fail, the mutex is active and we must stop the interpreter, since there is a mutex that hinders the current result from being true.

The mutex finding in Prolog looks like

```
check_for_mutex(Goal, _, _) :-
    \+mutex(Goal, M, _),
```

```
    \+mutex(M, Goal, _).
check_for_mutex(Goal, RuleBase, Proved) :-
    (
        mutex(Goal, M, C)
    ;
        mutex(M, Goal, C)
    ),
    (
        prove_condition(C, RuleBase, Proved)
        ->
        prove_mutex(M, RuleBase, Proved)
    ;
        true
    ).
```

The evaluation of the second element involved in the mutex is done by this code segment:

```
prove_mutex(M, RuleBase, Proved) :-
    (
        member((_, M), Proved) -> fail
    ;
        \+member((_,M), Proved),
        member((_, M, Body), RuleBase)
        ->
        (
            Body == [true] -> fail
        ;
            prove(Body, RuleBase, Proved, _) -> fail
        ;
            true
        )
    ;
        true
    ).
```

6.4 Problems and Solutions

No Direct Negation as Failure Support

Like the forward chaining interpreter, the backward chaining interpreter we designed does not support negation as failure. But it can make use of the transformation program of chapter 5 to transform a program with negation as failure into an equivalent one without it. Therefore, we must import the file *delfail.pl* and then call the predicate `preprocess_rules(Ruleset, Ruleset2)`. This transforms `Ruleset` (containing negation as failure) into `Ruleset2` (no negation as failure).

6.5 Evaluation

The performance of backward chaining on ordinary logic programs is excellent, since backward chaining works backwards from given goal looking for rules and facts to prove this goal. The computational complexity of such an interpreter would be $\mathcal{O}(n)$, with n being the number of rules in the program. That is exactly, what the normal Prolog interpreter does - so if it were not for courteous logic programs, we would not have to develop a backward chaining interpreter.

But we are interpreting courteous logic programs, which makes the proving process a lot more complicated. As said before in chapter 6.1, if we can prove that a predicate p is provable, we must assure that we cannot prove $\neg p$; otherwise, there is a conflict. To prove $\neg p$ it might be necessary to check all other rules in the program, what might be inefficient. It is, however, the only way to make sure that the answer set is consistent. This check itself has complexity $\mathcal{O}(n)$, which brings the computational complexity of the whole interpreter to $\mathcal{O}(2^n)$.

Things get even worse, when we include mutex detection in the complexity calculations. If there is a mutex, we must examine whether another element is true, which again takes time $\mathcal{O}(n)$. With this change, the overall complexity changes to $\mathcal{O}(3^n)$.

Using backward chaining to prove goals for courteous logic programs is very inefficient, since not only the rules the goal depends on are checked, but also opposing predicates and mutexes. It is probably faster to use forward chaining and then see whether the answer set contains the goal. Moreover, forward chaining has the advantage of finding all answers, not only one.

Backward chaining is fast if only one goal has to be proved and if the program contains few opposing predicates or mutexes. But on average, it is more efficient to use forward

chaining, although one has to keep in mind, that there are problems that can only be solved with backward chaining, for example recursive programs.

Finally, we ran the same 3 programs we used to measure the performance of the forward chaining interpreter on the backward chaining interpreter. Please remember, that we also ran 50 iterations, otherwise the time span used would be too short to measure for the smallest example. These are the results.

Finding the complete answer set for the mollusk example took an average of 55 ms, a little longer than for the forward chaining interpreter. Finding all answers to *ruleset4.pl* took 7372 ms and calculating the answer set for *ruleset5.pl* took 224212 ms. Calculating one element in the answer set took 0.12 ms in case of *example2.pl*, 0.79 ms in case of *ruleset4.pl*, and 3.3 ms in case of *ruleset5.pl*.

The difference between those two interpreters is not big for the mollusks example, since the code is too short to show a big difference. The processes around the benchmark (invoking the interpreter, collecting the rules, ...) are probably taking most of the time. When we, however, look at the other two examples, we can see the difference; backward chaining takes about 2 - 3 times longer to find the complete answer set.

Chapter 7

Conclusion

The present study is an approach to provide a Prolog implementation of a forward and a backward chaining interpreter for courteous logic programs. After a short introduction to courteous logic programming, it was defined how the language elements introduced by courteous logic programing should be represented. We then started to write the forward chaining interpreter for courteous logic programs. Forward chaining means, that the interpreter takes information it already knows (facts) as well as the rules of the program and then tries to derive new information from them.

In order to do this, we must check the body of each rule and see whether we can make it true using the information we currently know. If so, a new element can be derived and is saved to the list of known facts. The only problem with courteous logic programing is, that conflicts can arise, if the positive and negative version of a predicate are both true. In that case, we must look for an override. Overrides define, which of the elements has higher priority and which one can be deleted. If no prioritization is specified, the compiler takes the skeptical approach and none of the elements can be derived. The interpreter has to check for mutual exclusions between derived elements as well. If a matching mutex is found, none of the elements in the mutex can be true and both get deleted from the answer set.

Not all programs can be interpreted with forward chaining, for example recursive programs. Therefore, there was the need to develop the backward chaining interpreter. Backward chaining uses the top-down approach, rather than the bottom-up, applied by forward chaining. Given a predicate, the backward chaining interpreter tries to prove each element in the body of the rule. Proving these elements is done exactly in the same way, i.e. by trying to prove their body. This process keeps on until every element involved is proved.

To implement that behavior, we wrote a recursive predicate, which takes an element, tries to match it with a rule and then calls itself recursively for each element in the body of the rule. This is done until all elements involved are proved. It gets tricky, when we take into account, that rule heads can be negated as well. For every rule, there could be an opposing rule, but the opposing rule might not be needed at all by the backward chaining to find the result. Still, as long as it is provable, there will be a conflict. As a consequence, for every rule we try to prove, we also try to prove the opposing rule. If one is found, there is a conflict. It can be resolved by consulting the override rules. If the conflict cannot be resolved in favor of the original rule, the interpreter fails. Mutex's are handled in the same way. For each element, we check whether it is involved in a mutex. If the second element in that mutex is true, the interpreter fails.

Due to the introduction of logical negation in CLPs, CLPs now have two forms of negation: logical negation and negation as failure. The latter is hard to grasp for unexperienced users and used together with logical negation in one program, it can confuse even more. This paper therefore provides a method to remove negation as failure from a program, while the result stays the same.

To make use of this theory, we developed a precompiler, which takes a program with negation as failure and produces a program with the same output, but without negation as failure. This transformation is done by removing negation as failure from the rules and adding auxiliary rules, which use the skeptical behavior of the courteous logic interpreter to inhibit a rule from being true.

Outlook

Both interpreters lack support for Prolog system predicates. Prolog system predicates are commands like `is`, `==` or simple arithmetic operations (`+`, `-`, ...), as well as predicates imported from libraries, which, for example, support list manipulation (`member`, `append`, ...). Basic courteous logic programs can be used with the interpreter provided here, but before writing more complex programs, some more work needs to be done on the interpreters.

The interpreters performance currently is acceptable, but we believe that the algorithms to process the courteous logic rules can be optimized. We thought it is more important to have a working version of the interpreter first, even if it is not fast. Especially when the programs get large and include many conflicts and mutexes, the performance is bad. Since courteous logic programs probably will introduce lots of conflicts and mutexes, the performance of the interpreter will have to be optimized.

The original intention for this interpreter was to integrate it into ACE (Attempto Controlled English), but this task had to be dropped due to time restrictions. Nonetheless, this integration is interesting. ACE is a language especially designed to write specifications. It is a controlled natural language, i.e. a subset of English, which has a domain specific vocabulary and restricted grammar.[1] With this integration, it would be possible to express courteous rules in ACE, then transform those into CLPs, and interpret them using the interpreter introduced in this thesis.

[1]For more information on ACE see [Attempto].

Appendix A

Code of the Interpreters

A.1 clp.pl - Courteous Logic Operators

```
% This file contains all predicates that are needed to deal with
% courteous logic programs. Rules with empty label and facts without
% the [true] body part get converted through rules in this file.
% Also the operators that re needed for rules and classical negation are
% defined here.
%
% rule structure
% label :: Head <- [Body]. (rule)
% :: Head <- [Body].  (rule)
% label :: Fact <- [true].  (fact)
% :: Fact <- [true].  (fact)
% label:: Fact <- .  (fact)
% :: Fact <- .  (fact)

% operator for rule label
:- op(960, xfx, ::).
% operator for empty label rules
:- op(960, fx, ::).
% operator for head body separator
:- op(959, xfx, <-).
% operator for head body separator
:- op(959, xf, <-).
```

```
% operator for negation-as-failure
:- op(957, fx, ~).
% operator for not
:- op(956, fx, \-).
% operator for answerset
:- op(999, fx, \~).

:- multifile (::)/2.
:- multifile (::)/1.
:- multifile overrides/2.
:- multifile mutex/3.


% ::(+emptyLabel, -X)
%
% this generates a rule with label emptyLabel for a rule that has no
% label

emptyLabel :: X :- ::X.

% generates a fact without [true] in body

L :: Head <- [true] :- L :: Head <- .


% collectRules(-Rules)
%
% get all rules and facts, put them in a tupel (Label, Head, Body) and
% concatenate to a list

collect_rules(Rules) :-
    bagof((L,H,B), L :: H <- B, Rules).
```

```
% negate(+X, -NotX)
%
% negates the predicate with classical negation. positive -> negative and
% the other way round

negate(\-X, X).

negate(X, \-X) :-
    \+(X = (\-_)).

% overrides(_, +emptyLabel)
%
% global override giving all emptyLabel rules lowest priority

overrides(_, emptyLabel).
```

A.2 fwchn.pl - Forward Chaining Interpreter

```
% This file contains the forward chaining interpreter for courteous logic
% programs. It is started with the command \~X X is the variable to which
% the resulting answer set will be bound. Alternatively answerset(X) can
% be called to start up the interpreter.
% To use it with an own ruleset, it is necessary to include this file
% before the rules. This can be done by including it the :-[fwchn].
% command in your program file. After doing that, the above mentioned
% commands are available.


:- [clp].
:- [delfail].

% list_empty(+List)
%
% checks if a list is empty

list_empty([]).
```

```
% get_facts(+KnowledgeBase, -Facts)
%
% Finds all Facts in KnowledgeBase and returns them

get_facts(RuleSet, Facts) :-
    findall((L, H, [true]), member((L,H,[true]), RuleSet), TmpFacts),
    append_and_check_overrides([], TmpFacts, TFacts, _, DeleteList),
    cascade_delete_list(TFacts, DeleteList, Facts).

% get_rules(+KnowledgeBase, -Rules)
%
% finds all Rules in the KnowledgeBase and returns them

get_rules(RuleSet, Rules) :-
    findall(R, get_rule(RuleSet, R), Rules).

% get_rule(+RuleSet, -Rule)
%
% finds a single rule in the RuleSet.

get_rule(RuleSet, (L, H, B)) :-
    member((L, H, B), RuleSet),
    B \== [true].

% answerset(-Set)
%
% Returns all possible answers for the current program, ignores doubles

answerset(Aset) :-
    collect_rules(KB),
    preprocess_rules(KB, KBwoNAF),
    answerset(KBwoNAF, Answers),
    findall(A, member((_,A, _), Answers), AsetUnsorted),
    sort(AsetUnsorted, Aset).
```

```
% answerset(+RuleSet, -Answers)
%
% finds all Answers for RuleSet

answerset(RuleSet, Answers) :-
    get_facts(RuleSet, KB),
    get_rules(RuleSet, Rules),
    % filter only heads from KB, which is of form (Label, Head)
    findall(F, member((_,F, _), KB), LastWM),
    answerset(Rules, KB, LastWM, Answers).

% answerset(+RuleSet, +KnowledgeBase, +LastWorkingMemory, -Answers)
%
% finds all Answers for RuleSet with current KnowledgeBase
% LastWorkingMemory ist needed for optimization of rule selection.

answerset(_, KB, [], KB).

answerset(RuleSet, KB, LastWM, KB5) :-
    \+list_empty(LastWM),
    fire(RuleSet, KB, LastWM, WM),
    append_and_check_overrides(KB, WM, KB2, NewWM, DeleteList),
    cascade_delete_list(KB2, DeleteList, KB3),
    check_mutex(KB3, KB4),
    answerset(RuleSet, KB4, NewWM, KB5).

% fire(+RuleSet, +KnowledgeBase, -WorkingMemory)
%
% fire all rules in RuleSet that can be fired with the current
% KnowledgeBase and return result in WorkingMemory. For each rule, find
% all possible outcomes. based on the LastWorkingMemory a rule is
% qualified to fire or not.

fire([], _, _, []).
```

```
fire([(Label,Head,[true])|Rest], KB, LastWM, WM2) :-
    fire(Rest, KB, LastWM, WM),
    append([(Label,Head, [true])], WM, WM2).

fire([(Label,Head,Body)|Rest], KB, LastWM, WM3) :-
    check_rule_qualified(LastWM, Body),
    % get all possible instances of rule for the current KB
    findall((Label,Head, Body), fire_rule(Label, Head, Body, KB), WM),
    % fire rest of rules
    fire(Rest, KB, LastWM, WM2),
    append(WM, WM2, WM3).

fire([(_, _, Body)|Rest], KB, LastWM, WM) :-
    \+check_rule_qualified(LastWM, Body),
    fire(Rest, KB, LastWM, WM).

% check_rule_qualified(+LastWorkingMemory, +RuleBody)
%
% this is an optimization. It checks whether a rule is qualified to fire,
% by checking if an element of the LastWorkingMemory is contained in the
% RuleBody. Iff so, then the rule is allowed to fire.

check_rule_qualified([H|_],Body) :-
    copy_term(Body, Body2),
    % if H is found more than one times in Body2, then just doing a
    % member would cause a choicepointhere, using findall and then
    % checking if the list is not empty eliminates this choicepoint
    findall(H, member(H, Body2), HeadList),
    \+list_empty(HeadList).

check_rule_qualified([H|T], Body) :-
    copy_term(Body, Body2),
    \+member(H, Body2),
    check_rule_qualified(T, Body).
```

```
% fire_rule(+Label, +Head, +Body, +KnowledgeBase)
%
% Fires this rule with elements from the current KnowledgeBase
% This is done by recursivley stepping through each element
% and checking if it is satisfyable with the current KnowledgeBase

fire_rule(_, _, [true], _).

fire_rule(_, _, [], _).

fire_rule(Label, Head, [BodyElement|Rest], KB) :-
    member((_,BodyElement, _), KB),
    fire_rule(Label, Head, Rest, KB).

% \~(-X)
%
% Operator for answerset(X). Same as answerset(X).

\~(X) :-
    answerset(X).

% append_and_check_overrides(+KnowledgeBase, +WorkingMemory,
%                            -NewKnowledgeBase, -NewWorkingMemory,
%                            -DeleteList)
%
% Appends current WorkingMemory to current KnowledgeBase and thereby
% checkif there are any conflict. If conflicts are found, then resolve
% them.

append_and_check_overrides(KB, [], KB, [], []).

append_and_check_overrides(KB, [(Label, Head, Body)|Rest], NewKB,
                           NewWM, DL) :-
    append_and_check_overrides(KB, Rest, TmpKB, TmpWM, TmpDL),
    (
        % if Head already exists -> ignore
```

```
        member((_,Head, _), TmpKB)
        ->
        NewKB = TmpKB,
        NewWM = TmpWM,
        DL = TmpDL
    ;
        negate(Head, NegHead),
        (
            % if negated Head exists in KB, then a conflict has to be
            % resolved
            copy_term(TmpKB, TmpKB2),
            member((NegLabel, NegHead, NegBody), TmpKB2)
            ->
            resolve_conflict((Label, Head, Body), (NegLabel, NegHead,
                             NegBody), TmpKB, TmpWM, TmpDL, NewKB,
                             NewWM, DL)
        ;
            % else add Head to KB
            append([(Label, Head, Body)], TmpKB, NewKB),
            append([Head], TmpWM, NewWM),
            DL = TmpDL
        )
    ).

% resolve_conflict((+Label, +Head), (+ConflictingLabel,
%                  +ConflictingHead), +KnowledgeBase, +WorkingMemory,
%                  +DeleteList, -NewKnowledgeBase, -NewWorkingMemory,
%                  -NewDeleteList)
%
% Resolves the conflict between Head and ConflictingHead.
% Checks for overrides to see which rule has higher priority. If no
% overrides is found, return sceptical answer set

resolve_conflict((Label,_,_),(CLabel,_,_),KB,WM,DL,KB,WM,DL) :-
    CLabel \== emptyLabel,
    overrides(CLabel, Label).
```

```
resolve_conflict((Label, Head, Body), (CLabel, CHead, CBody), KB, WM,
                 DL, NewKB, NewWM, NewDL) :-
    Label \== emptyLabel,
    overrides(Label, CLabel),
    append([(CLabel, CHead, CBody)], DL, NewDL),
    append([(Label, Head, Body)], KB, NewKB),
    (
        member(CHead, WM)
        ->
        delete(WM, CHead, TWM),
        append([Head], TWM, NewWM)
    ;
        append([Head], WM, NewWM)
    ).

resolve_conflict((Label, Head, _), (CLabel, CHead, CBody), KB, WM, DL,
                 KB, NewWM, NewDL) :-
    (
        CLabel \== emptyLabel,
        overrides(Label, CLabel)
        ->
        fail
    ;
        Label \== emptyLabel,
        overrides(CLabel, Label)
        ->
        fail
    ;
        write('Conflict between '),
        write(Head), write(' and '), write(CHead), nl,
        write('Skeptical answer set returned.'), nl,
        append([(CLabel, CHead, CBody)], DL, NewDL),
        (
            member(CHead,WM)
            ->
```

```
            delete(WM, CHead, NewWM)
        ;
            NewWM = WM
        )
    ).

% cascade_delete(+KnowledgeBase, +DeleteItem, +NewKnowledgeBase)
%
% Deletes DeleteItem from KnowledgeBase using cascading delete. This
% means all items that depend on DeleteItem are removed.

cascade_delete(KB, DelItem, NewKB) :-
    delete(KB, DelItem, TKB),
    cascade_delete(TKB, DelItem, Deleted, TmpKB),
    cascade_delete_list(TmpKB, Deleted, NewKB).

% cascade_delete(+KnowledgeBase, +DeleteItem, -DeletedItems,
%                -NewKnowledgeBase)
%
% Delete all Items dependent on DeleteItem and save them to DeletedItems

cascade_delete([], _, [], []).

cascade_delete([(DLabel, DHead, DBody)|Rest], (Label, Head, Body),
                [(DLabel, DHead, DBody)|Deleted], NewKB) :-
    member(Head, DBody),
    cascade_delete(Rest, (Label, Head, Body), Deleted, NewKB).

cascade_delete([(KLabel, KHead, KBody)|Rest], (Label, Head, Body),
               Deleted, [(KLabel, KHead, KBody)|NewKB]) :-
    \+member(Head, KBody),
    cascade_delete(Rest, (Label, Head, Body), Deleted, NewKB).
```

```
% cascade_delete_list(+KnowledgeBase, +DeleteItemList, -NewKnowledgeBase)
%
% Cascade delete the first Item in DeleteItemList by calling
% cascade_delete and advance through the rest of the list

cascade_delete_list(KB, [], KB).

cascade_delete_list(TmpKB, [D|R], NewKB) :-
    cascade_delete(TmpKB, D, TmpKB2),
    cascade_delete_list(TmpKB2, R, NewKB).

% check_mutex(+KnowledgeBase, -MutexfreeKnowledgeBase)
%
% check KnowledgeBase for mutexes and if any are found, then the elements
% get removed

check_mutex(KB, KB2) :-
    findall((M1, M2, Condition), mutex(M1, M2, Condition), Mutex),
    check_mutex(Mutex, KB, KB2).

% check_mutex(+MutexList, +KnowledgeBase, -KnowledgeBaseWoMutex
%
% iterates through the mutex and if a mutex is "active" removes those
% two elements from the KnowledgeBase.

check_mutex([], KB, KB).

check_mutex([(MH1, MH2, Condition)|Rest], KB, KB3) :-
    member((ML1, MH1, MB1), KB),
    member((ML2, MH2, MB2), KB),
    check_condition(Condition, KB),
    cascade_delete_list(KB, [(ML1, MH1, MB1), (ML2, MH2, MB2)], KB2),
    check_mutex(Rest, KB2, KB3).
```

```
check_mutex([(MH1, MH2, Condition)|Rest], KB, KB2):-
    (
        member((_, MH1, _), KB),
        member((_, MH2, _), KB),
        check_condition(Condition, KB)
        ->
        fail
    ;
        check_mutex(Rest, KB, KB2)
    ).

% check_condition(+ConditionList, +KnowledgeBase)
%
% checks if condition is valid with current KnowledgeBase

check_condition([], _).

check_condition([Head|Tail], KB) :-
    member((_, Head, _), KB),
    check_condition(Tail, KB).
```

A.3 bwchn.pl - Backward Chaining Interpreter

```
% This file contains the code for the backward chaining algorithm. To
% use it import it as first thing you do in your ruleset file. importing
% can be done with :-[bwchn]. To start the interpreter call prove(Goal)
% or \~Goal. Goal can be either a term or a list of terms to prove.
% The rest works like the normal prolog interpreter. Results will be
% "yes" or "no" plus instanced variables. Alternative results can be
% called using ";".

:-[clp].
:-[delfail].
:-[stack].
```

```
% \~(+Goal)
%
% starts the interpreter to prove Goal

\~(G) :-
    prove(G).

% prove(+Goal)
%
% Starts the backward chaing interpreter to prove Goal. Goal can be
% either a term or a list of terms. The Goal(s) is then pushed on a stack
% and proving of that stack is started.

prove(Goal) :-
    \+is_list(Goal),
    collect_rules(RuleBase),
    preprocess_rules(RuleBase, RuleBase2),
    stack_push(Goal, [], GoalStack),
    prove(GoalStack, RuleBase2, [], _).

prove(Goals) :-
    is_list(Goals),
    collect_rules(RuleBase),
    preprocess_rules(RuleBase, RuleBase2),
    stack_push_many(Goals, [], GoalStack),
    prove(GoalStack, RuleBase2, [], _).

% prove(+GoalStack, +RuleBase, +Proved, -NewProved)
%
% Proves the GoalStack by trying to find a match for the first element of
% GoalStack in RuleBase. Proved is a list of terms proved so far. This is
% used for optimization of the proving of the opposing term. NewProved is
% the list of all proved elements during the prove process.

prove(GoalStack, RuleBase, Proved, NewProved) :-
    \+stack_is_empty(GoalStack),
```

```
    stack_pop(GoalStack, Goal, GoalStack2),
    member((_,Goal), Proved),
    prove(GoalStack2, RuleBase, Proved, NewProved).

prove(GoalStack, RuleBase, Proved, NewProved) :-
    \+stack_is_empty(GoalStack),
    stack_pop(GoalStack, Goal, GoalStack2),
    member((Label, Goal, Body), RuleBase),
    (
        Body == [true]
        ->
        GoalStack3 = GoalStack2
    ;
        Body \== [true]
        ->
        stack_push_many(Body, GoalStack2, GoalStack3)
    ),
    append(Proved, [(Label, Goal)], Proved2),
    prove(GoalStack3, RuleBase, Proved2, NewProved),
    check_for_opposer(Label, Goal, RuleBase, NewProved),
    check_for_mutex(Goal, RuleBase, NewProved).

prove(GoalStack, _, Proved, Proved) :-
    stack_is_empty(GoalStack).

% check_for_opposer(+Label, +Goal, +RuleBase, +NewProved)
%
% this predicate checks if there is a opposing goal for Goal in the
% RuleBase. If an opposer is found, then it tries to resolve the conflict
% by checking for overrides.

check_for_opposer(Label, Goal, RuleBase, Proved) :-
    negate(Goal, OppGoal),
    (
        member(OppGoal, Proved)
        ->
```

```
        fail
    ;
        \+member(OppGoal, Proved),
        findall(CLabel, prove_oppgoal(OppGoal, RuleBase, CLabel, Proved),
                CLabels)
        ->
        resolve_all_conflicts(Label, CLabels)
    ;
        true
    ).

% prove_oppgoal(+OpposingGoal, +RuleBase, -ConflictingLabel)
%
% Checks if OpposingGoal is true. If so, returns the Label of the
% Opposing Rule.

prove_oppgoal(OppGoal, RuleBase, CLabel, Proved) :-
    member((CLabel,OppGoal, Body), RuleBase),
    (
        Body == [true]
        ->
        true
    ;
        Body \== [true]
        ->
        prove(Body, RuleBase, Proved, _)
    ).

% resolve_all_conflicts(+Label, +ConflictingLabelList)
%
% Tries to resolve all conflicts between the elements of
% ConflictingLabelList and Label by looking for overrides rules.

resolve_all_conflicts(_, []).
```

```
resolve_all_conflicts(Label, [CLabel|Rest]) :-
    resolve_conflict(Label, CLabel),
    resolve_all_conflicts(Label, Rest).

% resolve_conflict(+Label, +ConflictingLabel)
%
% Is a helper Predicate for resolve_all_conflicts/2. It checks if there
% is a overrides rule to specify prioritization between Label and
% ConflictingLabel. If Label is of higher priority than ConflictingLabel,
% proceed. In any other case the predicate fails.

resolve_conflict(Label, CLabel) :-
    Label \== emptyLabel,
    overrides(Label, CLabel).

resolve_conflict(Label, CLabel) :-
    overrides(CLabel, Label),
    fail.

resolve_conflict(Label, CLabel) :-
    \+overrides(CLabel, Label),
    \+overrides(Label, CLabel),
    fail.

% check_for_mutex(+Goal, +RuleBase, +ProvedList)
%
% check if there is a active mutex for Goal.
% This is done by checking if there is a condidition involed in the mutex
% and if so, check if condition is provable and if so, check mutex.

check_for_mutex(Goal, _, _) :-
    \+mutex(Goal, M, _),
    \+mutex(M, Goal, _).
```

```
check_for_mutex(Goal, RuleBase, Proved) :-
    (
        mutex(Goal, M, C)
    ;
        mutex(M, Goal, C)
    ),
    (
        prove_condition(C, RuleBase, Proved)
        ->
        prove_mutex(M, RuleBase, Proved)
    ;
        true
    ).

% prove_condition(+Condition, +RuleBase, +Proved)
%
% Proves the Condition.

prove_condition([], _, _).

prove_condition(C, RuleBase, Proved) :-
    C \== [],
    prove(C, RuleBase, Proved, _).

% prove_mutex(+MutexElement, +RuleBase, +Proved)
%
% Proves the second element involved in a mutex.

prove_mutex(M, RuleBase, Proved) :-
    (
        member((_, M), Proved)
        ->
        fail
    ;
        \+member((_,M), Proved),
        member((_, M, Body), RuleBase)
```

```
        ->
        (
            Body == [true]
            ->
            fail
        ;
            prove(Body, RuleBase, Proved, _)
            ->
            fail
        ;
            true
        )
    ;
        true
    ).
```

A.4 delfail.pl - Precompiler to Remove Negation as Failure

```
% Ihis file contains the predicates necessary to transform program with
% negation as failure into a semantic equivalent program without
% negation as failure. This is done by removing the elements that contain
% negation as failure from a rules body and then replacing them with
% auxiliary predicates
%
% example: original rules looks like
% :: q <- [p, ~r].
% transformed into
% :: q <- [p, a].
% :: a <- .
% :: \-a <- [r].
%
% to use it, call preprocess_rules with the original ruleset as the first
% argument and the transformed rules will get bound to the second
% argument.
```

```
% preprocess_rules(+KnowledgeBase, -KnowledgeBaseWoNegationAsFailure)
%
% Takes the knowledge base and returns a KnowledgeBaseWoNegationAsFailure
% This is done by stepping through each rule and if the body contains
% negation as failure, then these predicates are removed from the body
% and a propositional predicate prop is added. Additionally a prositiv
% fact with the name of the propositional predicate is added and for each
% of the removed predicates another rule is added. The head of this rule
% is the negated version of the propositional predicate and the body is
% made of the positive version of the predicate, that was negated by
% negation as failure.

preprocess_rules(R, NR) :-
    preprocess_rules(R, NR, []).

preprocess_rules([], [], _).

preprocess_rules([Rule|Rest], NewRules, PropList) :-
    eliminate_negation_as_failure(Rule, ModRules, PropList, PropList2),
    preprocess_rules(Rest, NewRest, PropList2),
    append(ModRules, NewRest, NewRules).

% eliminate_negation_as_failure(+Rule, -RuleListWithAuxPred,
%                                +PropList, -NewPropList)
% this predicate removes the negation as failure terms from the body of
% the rule, adds the propositional predicate and also adds the auxiliary
% rules, that were generated.

eliminate_negation_as_failure((Label, Head, [true]), [(Label, Head,
                               [true])], PropList, PropList).

eliminate_negation_as_failure((Label, Head, Body), Rules3, PropList,
                              NewPropList) :-
    Body \== [true],
    bagof(F, member(~F, Body), Fails),
    remove_fails(Fails, Body, NoFailBody),
```

```
    generate_prop_pred_name(Head, PropList, NewPropList, PropName),
    % find all free variables in the failed items
    find_variables(Fails, [], VariableList),
    % generate propositional predicate with all free variables
    append([PropName], VariableList, P),
    PropPred =.. P,
    append(NoFailBody, [PropPred], FinalBody),
    % generate auxiliary rules
    add_aux_rules(Fails, PropPred, AuxRules),
    append([(Label, Head, FinalBody)], [], Rules2),
    append(Rules2, AuxRules, Rules3).

eliminate_negation_as_failure((Label, Head, Body), [(Label, Head, Body)],
                              PropList, PropList) :-
    Body \== [true],
    \+bagof(F, member(~F, Body), _).

% generate_prop_pred_name(Head, PropList, NewPropList, PropName)
%
% Generate the name of the propositional predicate. First get the name of
% the old head. Then add prop_ in front of it and a number at the end.
% The number is in case there are multiple predicates with the same name,
% that need to be replaced. The final result is then saved in PropName.

generate_prop_pred_name(Head, PropList, NewPropList, PropName) :-
    Head =.. L,
    nth0(0, L, Name),
    string_concat(prop_, Name, PN),
    (
        member((PN, Numb), PropList)
        ->
        Number is Numb + 1,
        delete(PropList, (PN, Numb), TmpPL),
        append([(PN, Number)], TmpPL, NewPropList)
    ;
        \+member((PN, _), PropList)
```

```
        ->
        Number = 0,
        append([(PN, Number)], PropList, NewPropList)
    ),
    string_concat(PN, Number, PN2),
    string_to_atom(PN2, PropName).

% remove_fails(+FailList, +Body, -BodyWithoutNegationAsFailure)
%
% removes the negation as failure elements from the Body

remove_fails([], B, B).

remove_fails([F|R], Body, Body3) :-
    delete(Body, ~F, Body2),
    remove_fails(R, Body2, Body3).

% add_aux_rules(+FailList, +PropPredicateName, -AuxRules)
%
% Generates the auxilliary rules for all the removes negation as failure
% predicates.

add_aux_rules([], PropName, [(aux, PropName2, [true])]) :-
    copy_term(PropName, PropName2).

add_aux_rules([Fail|Rest], PropName, [PropTerm|AuxRules]) :-
    copy_term((aux,\-PropName, [Fail]), PropTerm),
    add_aux_rules(Rest, PropName, AuxRules).

% find_variables(+PredList, +VariableList, -VariableList2)
%
% Find all free (unbound) variables in PredList and add them to the ones
% already in VariablesList and save the new list in VariableList2.

find_variables([], VL, VL).
```

```
find_variables([Fail|Rest], VL, VL4) :-
    Fail =.. Parts,
    scan_predicate_parts(Parts, VL2),
    append(VL, VL2, VL3),
    find_variables(Rest, VL3, VL4).

% scan_predicate_parts(+PredicateParts, -VariableList)
%
% Scan PredicateParts for unbound variables. Ignore the first element in
% the list, which is the predicate name.

scan_predicate_parts([_|Attr], VL) :-
    find_free_variables(Attr, VL).

% find_free_variables(+List, -VariableList)
%
% Find unbound variables in List and save them to VariableList.

find_free_variables([], []).

find_free_variables([X|R], [X|VL]) :-
    var(X),
    find_free_variables(R, VL).

find_free_variables([X|R], VL) :-
    \+var(X),
    find_free_variables(R, VL).
```

Appendix B

Code of the examples

B.1 example1.pl - Nixon Diamond

```
qua :: pacifist(X) <- [quaker(X)].
rep :: \-pacifist(X) <- [republican(X)].
:: quaker(nixon) <- [true].
:: republican(nixon) <- [true].

overrides(rep,qua).
```

B.2 example2.pl - Mollusks

```
m:: mollusk(X) <- [cephalopod(X)].
c:: cephalopod(X) <- [nautilus(X)].
mol :: shellbearer(X) <- [mollusk(X)].
cep :: \-shellbearer(X) <- [cephalopod(X)].
nau :: shellbearer(X) <- [nautilus(X)].

f1:: mollusk(molly) <- [true].
f2:: cephalopod(sophie) <- [true].
f3:: nautilus(natalie) <- [true].

overrides(nau, cep).
overrides(cep, mol).
overrides(nau, mol).
```

B.3 example3.pl - Personal E-Mail Agents

```
jun :: \-important(Msg) <- [from(Msg, X), retailer(X)].
del :: important(Msg) <- [from(Msg, X), awaitingDeliveryFrom(karen, X)].

overrides(del, jun).

:: awaitingDeliveryFrom(karen, parisCo) <- [true].
:: retailer(faveCo) <- [true].
:: retailer(babyCo) <- [true].
:: retailer(parisCo) <- [true].

:: from(110, parisCo) <- [true].
:: from(116, faveCo) <- [true].
:: from(211, babyCo) <- [true].

fav :: important(Msg) <- [from(Msg, faveCo)].

overrides(fav, jun).
```

B.4 example4.pl - Mail Importance: Family

```
clo :: important(Msg) <- [from(Msg, X), closeFamily(X, fred)].
dai :: \-important(Msg) <- [from(Msg, auntDaisy)].
eme :: important(Msg) <- [notificationOf(Msg, E), personalEmergency(E)].

overrides(dai, clo).
overrides(eme, dai).
overrides(eme, clo).

:: personalEmergency(S) <- [severeIllness(S, X), closeFamily(X, fred)].
:: closeFamily(betty, fred) <- [true].
:: closeFamily(auntDaisy, fred) <- [true].

:: from(item19, betty) <- [true].
:: from(item20, auntDaisy) <- [true].
```

```
:: from(item115, auntDaisy) <- [true].
:: notificationOf(item115, sit79) <- [true].
:: severeIllness(sit79, auntDaisy) <- [true].
```

B.5 tweety.pl - Tweety the Penguin

```
fly :: fly(X) <- [bird(X)].
nfly :: \-fly(X) <- [penguin(X)].
peng :: penguin(X) <- [walkslikepeng(X)].
npeng :: \-penguin(X) <- [\-flatfeet(X)].
bird :: bird(X) <- [penguin(X)].
bird :: bird(X) <- [wounded_bird(X)].

:: bird(tweety) <- .
:: walkslikepeng(tweety) <- .
:: \-flatfeet(tweety) <- .

:: penguin(sam) <- .

:: wounded_bird(john) <- .

overrides(npeng,peng).
overrides(nfly,fly).
mutex(fly(X), wounded_bird(X), [bird(X)]).
```

B.6 ruleset4.pl - Benchmark Example One

```
a :: a(0) <- .
a :: a(1) <- .
a :: a(3) <- .
b :: b(1) <- .
b :: b(4) <- .
c :: c(a) <- .
c :: c(b) <- .
c :: c(e) <- .
```

```
d :: d(X,Y) <- [a(X), b(Y)].
d :: d(X,Y) <- [a(X), a(Y)].
e :: e(X,Y,Z,A) <- [b(Y), c(Z), d(X, A)].
ne :: \-e(X,Y,Z,A) <- [b(X), c(Y), a(A), d(A,Z)].
f :: f(X) <- [d(X, 1)].
nf :: \-f(X) <- [a(X)].
g :: g(X,Y,Z) <- [\-f(X), e(X,Z,Y,X)].
n :: n(s(s(s(s(s(1)))))) <- .
overrides(nf, f).
```

B.7 ruleset5.pl - Benchmark Example Two

```
a :: a(0) <- .
a :: a(1) <- .
a :: a(2) <- .
a :: a(3) <- .
b :: b(1) <- .
b :: b(4) <- .
b :: b(4) <- .
b :: b(7) <- .
b :: b(3) <- .
b :: b(9) <- .
c :: c(a) <- .
c :: c(b) <- .
c :: c(c) <- .
c :: c(e) <- .
d :: d(X,Y) <- [a(X),b(Y)].
d :: d(X,Y) <- [a(X),a(Y)].
e :: e(X,Y,Z,A) <- [b(Y),c(Z),d(X,A)].
ne :: \-e(X,Y,Z,A) <- [b(X),c(Y),a(A),d(A,Z)].
f :: f(X) <- [d(X,1)].
nf :: \-f(X) <- [a(X)].
g :: g(X,Y,Z) <- [\-f(X),e(X,Z,Y,X)].
j :: j(X,Y,Z) <- [\-f(X),g(X,Y,Z)].
nj :: \-j(X,a,X) <- .
overrides(nf, f).
```

Appendix C

Code of Helper Applications

C.1 stack.pl - Stack Operations

```
% This file contains the code for basic stack operations.

stack_is_empty([]).

stack_push(E, Stack, [E|Stack]) :-
    \+is_list(E).

stack_pop(Stack, E, NewStack) :-
    \+stack_is_empty(Stack),
    [E|NewStack] = Stack.

stack_push_many([], Stack, Stack).

stack_push_many([H|T], Stack, [H|NewStack]) :-
    stack_push_many(T, Stack, NewStack).

stack_pop_many(0, Stack, [], Stack).

stack_pop_many(Num, Stack, [E|List], NewStack) :-
    Num1 is Num - 1,
    stack_pop(Stack, E, TmpStack),
    stack_pop_many(Num1, TmpStack, List, NewStack).
```

C.2 benchfw.pl - Forward Chaining Benchmark

```
% This is the program used to benchmark the forward chaining interpreter.
%
% First load a courteous logic program. Then call ":- lots."
% Now the program starts to seek all solutions for the program loaded
% before. This is repeated 50 times. This number can be changed by
% editing the eg_count predicate.
%
% Call ":- do(Num)" to repeat this "Num" number of times.

eg_count(50).

get_cpu_time(T) :- statistics(runtime,[T,_]).

lots :-
    eg_count(Count),
    bench(Count),
    fail.
lots.

bench(Count) :-
    get_cpu_time(T1),
    dobench(Count),
    get_cpu_time(T2),
    report(Count,T1,T2).

dobench(Count) :-
    repeat(Count),
    \~_,
    fail.
dobench(_).

repeat(_N).
repeat(N) :-
    N > 1,
```

```
    N1 is N-1,
    repeat(N1).
report(Count,T1,T2) :-
    Time is T2-T1,
    write(Count),
    write(' iterations taking '), write(Time),
    write(' msecs'),
    nl.

do(0).
do(N) :-
    lots,
    N1 is N - 1,
    do(N1).
```

C.3 benchbw.pl - Forward Chaining Benchmark

```
% This is the program used to benchmark the backward chaining
% interpreter.
%
% First load a courteous logic program. Then call ":- lots."
% Now the program starts to seek all solutions for the program loaded
% before. To find all solutions with the backward chaining interpreter,
% setof is used. This is done 50 times. This number can be changed by
% editing the eg_count predicate.
%
% Call ":- do(Num)" to repeat this Num number of times.

eg_count(50).

get_cpu_time(T) :- statistics(runtime,[T,_]).

lots :-
    eg_count(Count),
    bench(Count),
    fail.
```

```
lots.

bench(Count) :-
    get_cpu_time(T1),
    dobench(Count),
    get_cpu_time(T2),
    report(Count,T1,T2).

dobench(Count) :-
    repeat(Count),
    setof(X, \~X, _),
    fail.
dobench(_).

repeat(_N).
repeat(N) :-
    N > 1,
    N1 is N-1,
    repeat(N1).
report(Count,T1,T2) :-
    Time is T2-T1,
    write(Count),
    write(' iterations taking '), write(Time),
    write(' msecs'),
    nl.

do(0).
do(N) :-
    lots,
    N1 is N - 1,
    do(N1).
```

Bibliography

[Antoniou98]

Grigoris Antoniou, Michael J. Maher, David Billington. *Defeasible logic versus Logic Programming without Negation as Failure.* In The Journal of Logic Programming. November 1998.

[Baral94]

Chitta Baral, Michael Gelfond. *Logic Programming and Knowledge Representation.* January 1994.

[Gelfond91]

Michael Gelfond, Vladimir Lifschitz. *Classical Negation in Logic Programs and Disjunctive Databases.* 1991.

[Grosof97]

Benjamin N. Grosof. *Courteous logic programs: Prioritized conflict handling for rules.* Technical report, IBM T.J. Watson Research Center, IBM Research Report RC 20836. December 1997.

[Grosof99]

Benjamin N. Grosof. *A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Progams.* IBM T.J. Watson Research Center, Supplementary Update Follow-On to IBM Research Report RC 21472. July 1999.

[Reiter78]

Raymond Reiter. *On closed world data bases.* In Herve Gallaire and Jack Minker, editors, Logic and Data Bases, pages 119-140. Plenum Press. New York, 1978.

[Russel03]

Stuart J. Russel, Peter Norvig. *Artificial intelligence: a modern approach.* 2nd ed. Prentice Hall/Pearson Education. 2003.

[Attempto]
http://www.ifi.unizh.ch/attempto/

[CommonRules]
http://ebusiness.mit.edu/bgrosof/#CommonRules

[Gentoo]
http://www.gentoo.org

[OWL]
http://kaon.semanticweb.org/owl

[SWEET]
http://ebusiness.mit.edu/bgrosof/#SweetSoftware

[SWI]
http://www.swi-prolog.org

[WikipFwChn]
http://en.wikipedia.org/wiki/Forward_chaining

[WikipBwChn]
http://en.wikipedia.org/wiki/Backward_chaining