Artificial Intelligence Institute
Department of Computer Science
Dresden University of Technology

Großer Beleg

Extracting Logic Programs from Artificial Neural Networks

Jens Lehmann

February 22, 2005

Supervisors: Prof. Steffen Hölldobler
Dr. Pascal Hitzler
M.Sc. Sebastian Bader

Task and General Information

student:	name:	Jens Lehmann
	date and place of birth:	March 29th 1982, Meissen
	matr. nr.:	2851281

topic: The research field of neurosymbolic integration aims at combining the advantages of neural networks and logic programs.
One of the fundamental questions is how knowledge in form of logic programs can be extracted from neural networks. It should be evaluated if techniques of inductive logic programming (ILP) are suited for this task.
Additionally it should be investigated if there are good special purpose algorithms for the base case of propositional logic.

goals:

- evaluating existing ILP programs for the extraction of a logic program from a given neural network representing an immediate consequence operator
- find satisfying extraction methods for the special case of propositional logic programs

Abstract

This document is essentially divided in two parts, where different methods are presented for extracting knowledge from an aritificial neural network representing an immediate consequence operator.

In the first part we investigate the relationship between neurosymbolic integration (in particular the extraction of a logic program from a neural network) and inductive logic programming from a practical point of view. After a general introduction to the foundations of ILP, the task of extraction of a neural network is reformulated to fit the problem setting of ILP. We then practically test a variety of different programs and evaluate them.

The second part of the document builds up a theoretical foundation for the special case of extracting propositional logic programs. We give algorithms for definite as well as normal propositional logic programs. Several theoretical results are presented, difficulties and possible solutions are observed.

Contents

1 Introduction

Artificial neural networks and logic programming are both well established areas of artificial intelligence (see [RN03] for a general introduction). They both have different advantages and disadvantages. Neural networks are massively parallel, flexible, robust, and have well known field-tested learning algorithms ([Roj02] provides an introduction to neural networks). However the learned knowledge is not directly accessable. Logic programs, on the other hand, are easy to understand and have approved deduction algorithms. Neurosymbolic integration is a field, which aims to combine both paradigms (see e.g. [HK94], [HKS99], [BHH04]). Naturally, one of the main goals of neurosymbolic integration is to be able to understand neural networks by extracting knowledge in form of logic programs. This could help to understand these networks, which again may also lead to a better understanding of real (non-artificial) neural networks—like the human mind. It may also be used for massively parallel deduction systems [HK94] or for creating a learning cycle as illustrated in Figure 1 (a similar scheme can be found in [dGBG01]). Currently there is a lot of interest in neurosymbolic integration, but the whole field has still to be applied to practical problems.

To build a bridge between logic programming and neural networks two directions have to be researched: The first one is to build up a neural network from a logic program and the second is to extract knowledge in form of logic programs from neural networks. This document only observes the latter case. We restrict ourselves to networks, which represent an immediate consequence operator of an (unknown) logic program. This means the input I and output O of the network represent interpretations with $O = T_P(I)$. This idea was introduced in [HK94] for the case of propositional logic. It is not yet clear how to obtain networks computing T_P for normal first order logic programs (this is discussed in [HKS99], [BH04], [BHH04]). By restricting ourselves to networks, which represent an immediate consequence operator, we can abstract away from the details of the neural network we are viewing and only consider it as a black box. This is a fundamental difference to extraction methods, which rely on the inner structure of the network (see e.g. [dGBG01] for such systems). This also means that it may be possible to apply the results and observations made in this document to other areas of artificial intelligence (and logic programming in particular) although the motivation behind it clearly comes from the background of neurosymbolic integration.

The document is roughly divided into two parts with different ideas to solve the problem of knowledge extraction. The first part evaluates the use of inductive logic programming (ILP), which will be explained later, and the second part shows some results of directly trying to solve the problem. To make the notions used in these two parts clear we introduce some necessary preliminaries.

2 Preliminaries

Some knowledge in logic and logic programming is necessary to understand this document. The logical notions and definitions will only be introduced in a very compact

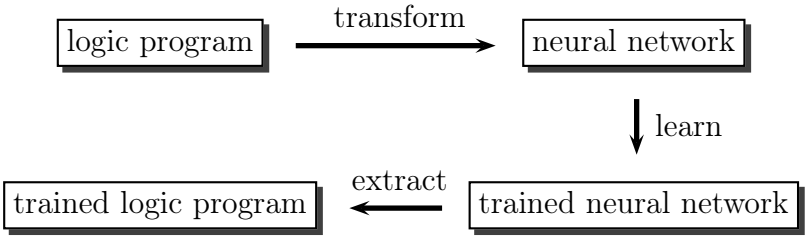


Figure 1: learning cycle

way. Every standard book about logic and logic programming like e.g. [Apt97] will cover this in more detail. The foundations of inductive logic programming will be discussed seperately in Section 3.

An *alphabet* of predicate logic contains a set $\mathcal{R}$ of relation (respectively predicate) symbols, a set $\mathcal{F}$ of function symbols, a set $\mathcal{C}$ of constants and a set $\mathcal{V}$ of variables. A *term* is recursively defined as a variable $v \in \mathcal{V}$, a constant $c \in \mathcal{C}$ or an expression of the form $f(t_1, \ldots, t_n)$ with $n \geq 1$ where all t_i $(1 \leq i \leq n)$ are terms and $f \in \mathcal{F}$ is a function symbol of *arity* n. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$ with $n \geq 0$ where all t_i $(1 \leq i \leq n)$ are terms and $p \in \mathcal{R}$ is a predicate symbol of arity n. A *literal* is an atom or a negated atom (negation is denoted by $\neg$). $L_1 \wedge L_2$ is a conjunction and $L_1 \vee L_2$ is a disjunction of two literals L_1 and L_2. A *clause* is a finite disjunction of literals. A *Horn clause* is a clause with at most one positive literal.

A *logic program* P is a finite set of *Horn clauses* of the form $H \leftarrow L_1, \ldots, L_n$ with $n \geq 0$ where all L_i $(0 \leq i \leq n)$ are literals and H is an atom. If $n = 0$ then the clause is called a *fact*. H is called *head* of the clause and $L_1, \ldots, L_n$ body of the clause. *Definite* clauses are clauses containing only atoms in their body. A *definite program* is a finite set of definite clauses.

A term, atom, literal, clause or program is *ground*, if it does not contain any variable. The *term universe* (also called *Herbrand universe*) U_P of a logic program P is the set of all ground terms. The Herbrand base B_P of P is the set of all ground atoms. A *Herbrand interpretation* I (or just called interpretation in this paper) is a subset of B_P. All ground atoms in I are assumed to be mapped to `true`. All ground atoms in $B_P \setminus I$ are assumed to be mapped to `false`. An interpretation is extended to literals, clauses and programs in the usual way. The *space of interpretations* I_P is the power set of B_P. A *model* M of an atom A (denoted $M \models A$) is an interpretation, which maps A to `true`. This is extended to literals, clauses and programs as usual.

A *substitution* θ is a partial function mapping variables to terms where a variable cannot be mapped to itself. θ is denoted as $\{x_1/t_1, \ldots, x_n/t_n\}$ with $n \geq 0$. A substitution is empty if $n = 0$ and ground if all t_i $(1 \leq i \leq n)$ are ground terms. Substitutions can be applied to terms, atoms, literals and clauses as usual. A *ground instance* of a clause is the result of the application of a ground substitution to a clause. A Horn clause C_1 of the form $H_1 \leftarrow L_1, \ldots, L_m$ *subsumes* a Horn clause C_2 of the form $H_2 \leftarrow M_1, \ldots, M_n$ if there is a substitution δ with $H_1\delta = H_2$ and $\{L_1\delta, \ldots, L_m\delta\} \subseteq \{M_1, \ldots, M_n\}$.

For this paper we will often need the notion of a T_P operator, also called the immediate consequence operator (often abbreviated as "operator" in this paper). We define operators as follows:

Definition 2.1 (T_P operator)
T_P is a mapping from interpretations to interpretations defined in the following way for an interpretation I and a program P:

$$T_P(I) := \{H \mid H \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m \in ground(P), n \geq 0, m \geq 0, \\ \{B_1, \ldots, B_n\} \subseteq I, \{C_1, \ldots, C_n\} \cap I = \emptyset\}$$

□

Two operators T_P for a program P and T_Q for a program Q are *equal* (denoted $T_P = T_Q$) if for all possible interpretations we have $T_P(I) = T_Q(I)$.

3 Heuristic Approach using ILP

3.1 Introduction to ILP

The task of learning a theory from examples is commonly called *induction*. It has been an interesting domain for scientists (mostly in philosophy) for a long time. It recently became more popular since more and more powerful computers are available. There are several methods of inductive learning, e.g. decision tree learning. Inductive logic programming tries to combine learning methods with the power of first order logic. It has gained popularity because of its rigorous approach to the problem of inductive learning and the use of powerful algorithms already known from logic.

The name of the field was invented in 1990 by Stephen Muggleton ([NCdW97]). ILP was applied to a great variety of practical problems. In 2001 automatically discovered rules of protein folding where important enough to be published in a scientific biology magazine. In this case ILP outperformed approaches like neural nets and decision trees. ILP is used in medicine to predict the therapeutic efficacy of drugs from their molecular structure. It is also used in natural language processing (these application areas are listed in [RN03], and [MR94] mentions a lot more of them).

3.2 The Normal Problem Setting in ILP

Before having a close look at the problem, which we want to solve, we describe the problem setting of ILP in more detail. As already mentioned, induction means to learn from examples. The result of this process is called a *theory*. Often the learning process is not done from scratch, but with some amount of *background knowledge* B. ILP takes this into account as we will see later. Moreover there are also two kinds of examples: *positive examples* E^+ and *negative examples* E^-. Positive examples are true and negative examples are false.

How are the background knowledge and the examples specified? Here logic programming comes into play. The background knowledge as well as the examples are usually

finite sets of clauses. Because of the popularity of Prolog a lot of ILP programs use a very similar syntax.

We want to introduce at least some formal definitions although this section is intended to be mostly of practical value. The following definitions and propositions are taken from [NCdW97].

Definition 3.1
A theory is a finite set of clauses. □

Definition 3.2
If $\Sigma = \{C_1, C_2, \dots\}$ is a set of clauses, then we use $\overline{\Sigma}$ to denote $\{\neg C_1, \neg C_2, \dots\}$. □

This now allows us to define what properties a good theory should have.

Definition 3.3 (correct theories)
Let Σ be a theory and E^+ and E^- be sets of clauses. Σ is *complete* with respect to E^+, if $\Sigma \models E^+$. Σ is *consistent* with respect to E^- if $\Sigma \cup \overline{E^-}$ is satisfiable. Σ is *correct* with respect to E^+ and E^- if Σ is complete with respect to E^+ and consistent with respect to E^-. □

Completeness means that we want a theory from which we can deduce all positive examples. Consistency is harder to understand. For consistency of Σ it is necessary that Σ does not imply a clause in E^-. However this is not sufficient. Let $\Sigma = \{P(a) \vee P(b)\}$ and $E^- = \{P(a), P(b)\}$. Then we have $\Sigma \not\models P(a)$ and $\Sigma \not\models P(b)$. But still Σ is not an intended theory, because it is clearly false ($P(a)$ and $P(b)$ are false, so their conjunction is false). The following observation makes this clearer:

Proposition 3.4 (consistency)
Let Σ be a theory and $E^- = \{e_1, e_2, \dots\}$ be a set of clauses. Then Σ is not consistent with respect to E^- if and only if there are $i_1, \dots, i_n$ such that $\Sigma \models e_{i_1} \vee \dots \vee e_{i_n}$.

A common setting in ILP is a restriction to definite programs as theories and ground atoms as examples ([NCdW97]). For this scenario we can easier understand consistency as shown in the next proposition.

Proposition 3.5
Let P be a definite program and E^- a set of ground atoms. Then P is consistent with respect to E^- if and only if $P \not\models e$ for every $e \in E^-$.

The learning problem in ILP is now formally defined as the problem of finding a correct theory with respect to the given background knowledge and examples. This rigorous definition of the problem is a reason for the recent success of ILP and has led to a rich theory.

In the sequel we give further important notions to describe the theories, which are induced by ILP systems.

Definition 3.6
Let Σ be a theory and E^+ and E^- be sets of clauses. Σ is *too strong* with respect to E^-, if Σ is not consistent with respect to E^-. Σ is *too weak* with respect to E^+, if Σ is not complete with respect to E^+.

Σ is *overly general* with respect to E^+ and E^-, if Σ is complete with respect to E^+, but not consistent with respect to E^-. Σ is *overly specific* with respect to E^+ and E^-, if Σ is consistent with respect to E^- but not complete with respect to E^+. □

Note that a theory can be too strong and too weak. We illustrate this by an example.

Example 3.7 (too strong, too weak, overly general, overly specific, correct)
Suppose the following is given:

- $B = \emptyset$
- $E^+ = \{odd(s(0)); odd(s(s(s(0))))\}$
- $E^- = \{odd(0); odd(s(s(0)))\}$
- $\Sigma_1 = \{odd(s(s(x)))\}$
- $\Sigma_2 = \{odd(s(0))\}$
- $\Sigma_3 = \{odd(s(0)); odd(s(s(s(0))))\}$
- $\Sigma_4 = \{odd(s(0)); odd(s(s(x))) \leftarrow odd(x)\}$

Σ_1 is too weak (with respect to E^+), because $\Sigma_1 \not\models odd(s(0))$. Σ_1 is also too strong (with respect to E^-), because $\Sigma_1 \models odd(s(s(0)))$. Because Σ_1 is too strong and too weak it is not overly specific and not overly general.

Σ_2 is too weak, because $\Sigma_2 \not\models odd(s(s(s(0))))$, and it is not too strong. Thus Σ_2 is overly specific.

Σ_3 and Σ_4 are correct. □

Most ILP systems start with a theory Σ and test, wether the theory is too weak or too strong. If it is too weak the theory is generalized and if it is too strong it is specialized. This process continues until a theory is found, which is neither too strong nor too weak. Such a theory is obviously correct by Definition 3.6.

This means that the two basic steps performed by ILP systems are *specialization* and *generalization*. Specialization is the search for a theory, which is consistent with (together with the background knowledge) with respect to the negative examples. Generalization means finding a theory, which (together with the background knowledge) implies all the positive examples. We will discuss different strategies to perform the specialization and generalization steps.

3.3 ILP Methods

We will now explain two ILP methods, namely model inference and inverse resolution. Please note that there are many other ILP methods, so our intention is only to give a general idea of ILP techniques. A more detailed overview can be found in [MR94] and [NCdW97].

3.3.1 Model Inference

Model inference is a prominent problem in ILP. Introducing model inference in detail is out of the scope of this paper, so we will introduce it in an informal way. Model inference was first described by Shapiro [Sha91].

For this problem we assume an *oracle* exists. An oracle can answer if a formula is correct. For instance a scientist can be an oracle. A justification for considering such an oracle as given is that a scientist may be able to answer specific instances of problems, but does not know the general rules.

For the model inference algorithm we need two additional tools: an algorithm for finding false clauses in a theory and a refinement operator. We will explain these in turn.

The algorithm for finding a false clause is called the *Backtracing algorithm*. It corresponds to the specialization step in the ILP setting, because we will use it to remove clauses from a theory (which obviously weakens it). The algorithm uses the oracle to find a false clause (theory is too strong). It takes as input a negative example, which is true according to a theory. The algorithm finds a clause in the theory, which is "responsible" for the wrong classification of the example. (We will not go into the details of the algorithm.)

The second tool is a (downward) refinement operator. It is used to strengthen a theory, which is too weak. For this, weaker versions of clauses, which were previously deleted by the Backtracing algorithm, are added. As a simple example $odd(s(s(x))) \leftarrow odd(x)$ is a weaker version of $odd(s(s(x)))$. The intuition behind this is the hope that the weaker version will not make the theory too strong. A refinement operator takes an arbitrary element of the theory as input and returns a set of possible specializations of this element. The refinement operator must be specified along with the examples for the learning algorithm.

With this in mind we can now present the model inference algorithm[1]:

Algorithm 3.8 (model inference)
initalize: set $\Sigma = \{\square\}$
repeat:
    read an example
    repeat:
        while Σ is too strong find a false clause and delete it from Σ
        while Σ is too weak add refinements of previously deleted clauses
    until Σ is correct with respect to the facts read so far □

[1] $\square$ denotes the empty clause

To illustrate the algorithm we will use a slightly changed version of an example from [NCdW97]. We suppose that the examples for the algorithm are ground atoms given in the listed order: $even(0)$ (positive), $even(s(0))$ (negative), $even(s^2(0))$ [2] (positive), $even(s^3(0))$ (negative). In fact Shapiro considers the case of a complete enumeration of all examples, but for the model inference algorithm to terminate we can only read a finite number of examples and must stop at some example (for this the author would suggest using a heuristic like: stop if the theory has not changed while reading the n previous examples with e.g. $n = 10$).

Let the refinement operator φ be as follows:

$$\varphi(C) = \begin{cases} \{even(x)\} & \text{if } C = \square \\ \{even(s^{n+1}(x)), (even(s^n(x)) \leftarrow even(x)), & \\ even(s^n(0))\} & \text{if } C = even(s^n(x)) \\ \emptyset & \text{otherwise} \end{cases}$$

We will now go through the algorithm step by step.

1. Set Σ to $\{\square\}$.
2. Read $even(0)$.
3. Σ is neither too strong nor too weak for the examples read so far.
4. Read $even(s(0))$.
5. Σ is too strong, because $\Sigma \models even(s(0))$. The false clause $\square$ is deleted, so $\Sigma = \emptyset$.
6. Σ is now too weak, because $\Sigma \not\models even(0)$. Add $\varphi(\square) = \{even(x)\}$ to Σ.
7. Σ is now too strong. Delete the false clause $even(x)$. $\Sigma = \emptyset$ again.
8. Σ is too weak. Add $\varphi(even(x)) = \{even(s(x)), (even(x) \leftarrow even(x)), even(0)\}$ to Σ.
9. Σ is too strong. Delete the false clause $even(s(x))$ and the tautology $even(x) \leftarrow even(x)$. Then $\Sigma = \{even(0)\}$ is neither too strong nor too weak with respect to the examples read so far.
10. Read $even(s^2(0))$.
11. Σ is too weak. Use the refinement operator again and add $\varphi(even(s(x)) = \{even(s^2(x)), (even(s(x)) \leftarrow even(x)), even(s(0))\}$ to Σ.
12. Σ is too strong. Delete $even(s(x)) \leftarrow even(x)$ and $even(s(0))$. Then $\Sigma = \{even(s^2(x)), even(0)\}$ is neither too strong nor too weak.

[2] $s^n(0)$ abbreviates applying s n-times to 0, i.e. $s^2(0)$ stands for $s(s(0))$

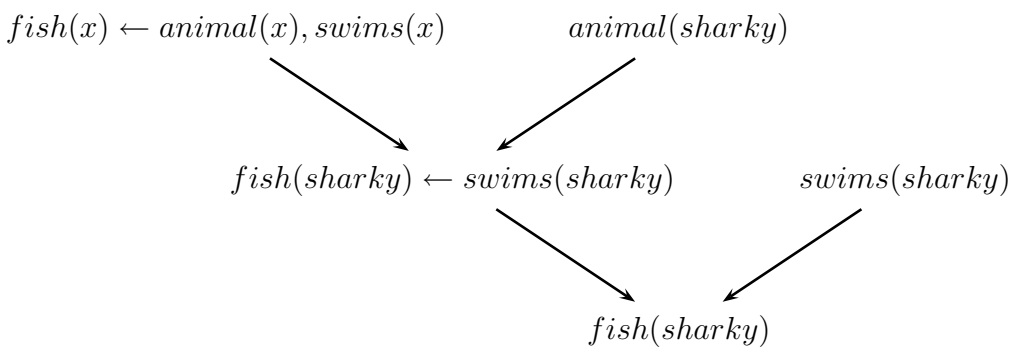


Figure 2: the V operator

13. Read $even(s^3(0))$.
14. Σ is too strong. Delete $even(s^3(0))$.
15. $\Sigma = \{even(0)\}$ is too weak. Add $\varphi(even(s^2(x)) = \{even(s^3(x)), (even(s^2(x)) \leftarrow even(x)), even(s^2(0))\}$ to Σ.
16. Σ is too strong. Delete $even(s^3(0))$.
17. $\Sigma = \{even(0), even(s^2(0)), (even(s^2(x)) \leftarrow even(x))\}$.

3.3.2 Inverse Resolution

A famous ILP method introduced by Muggleton and Bluntine [MB88] is *inverse resolution*. Since induction can be seen as the inverse of resolution it seems to be a good idea to revert the resolution process. For this two operators were introduced: the W and the V operator. We will introduce those operators following [MR94] and [NCdW97]. Again we will leave out the formal details.

The V operator takes as input clauses C_1 and R and finds a clause C_2 such that R is an instance of a resolvent of C_1 and C_2. It is called operator, because if you draw a resolution of two clauses to a resolvent this looks like a "V". Suppose we have the theory $\Sigma = \{animal(sharky); swims(sharky)\}$ and get the new example $fish(sharky)$. Our theory is too weak, because it does not imply the new example, so we must generalize it. Figure 2 illustrates this (SLD resolution is used there). In the first step R_1 is $fish(sharky)$ (this is what our new theory should imply) and C_1 is $swims(sharky)$ (this is an arbitrary choice of a clause in our theory or the background knowledge). Now we could for instance choose $C_2 = fish(sharky) \leftarrow swims(sharky)$. This choice is again not deterministic. In general inverse resolution gives much freedom for different choices. We could now add C_2 to Σ or do another inverse resolution step as shown in Figure 2. If we do this the new theory we get is $animal(sharky); swims(sharky); fish(x) \leftarrow animal(x), swims(x)$.

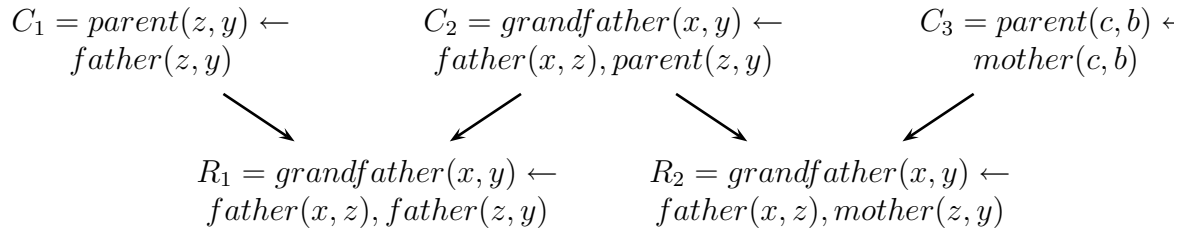


Figure 3: the W operator

The W operator is a combination of two V operators. It takes clauses R_1 and R_2 as input and finds clauses C_1, C_2 and C_3 such that R_1 is an instance of a resolvent of C_1 and C_2 and R_2 is an instance of a resolvent of C_2 and C_3. Figure 3 illustrates this case. The W operator forms a "W" in the graphical representation. Hence its name. In the figure we have two clauses R_1 and R_2, which describe the predicate $grandfather$. However the definition of $grandfather$ could be written more succinct as one clause if we had the predicate $parent$. During resolution such a predicate can disappear, so it is natural that inverse resolution should invent new predicates if appropriate. The W operator detects similarities between R_1 and R_2. After that a clause C_2 is created, which is intented to be more general than R_1 and R_2 and can contain new predicates. Now the V operator can be used for R_1 and C_2 respectively R_2 and C_2.

3.4 Analyzing the Task

We want to observe if ILP systems are a good way of extracting knowledge from neural networks. The neural networks we investigate shall represent a T_P operator.

- input: a set of pairs $(I, T_P(I))$, where I is a Herbrand interpretation
- output: a logic program P which (approximately) has operator T_P

Of course we practically do not really need a neural network for this task, but can instead consider a logic program and a function, which computes T_P for this program, where we are only allowed to use this function. With this method we can compare the output of the ILP algorithms with the original program and draw conclusions.

It is also important to notice that for this task we do not necessarily consider the whole T_P operator as given, but only a set of pairs $(I, T_P(I))$. These pairs serve as examples for the ILP algorithms. Obviously for the practical tests there can only be finitely many such examples. This means for the case where B_P is infinite we can never use the whole T_P operator as pairwise input for the ILP algorithms.

The whole setup is illustrated in Figure 4. The program P and the network are in some sense "equal" (shown by the doubleline and the dotted box in the figure) because we consider the network as black box and therefore it is only another way to view the program.

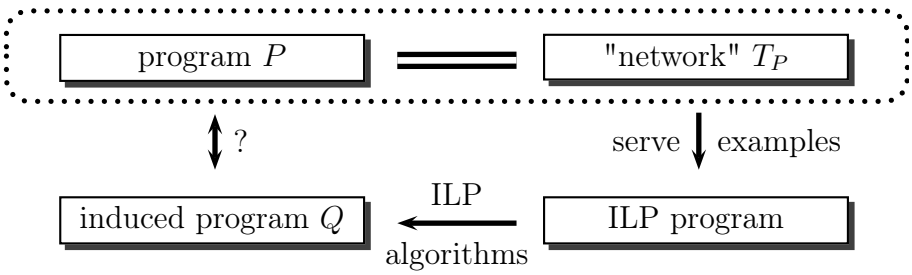


Figure 4: test setup

ILP systems learn theories from background knowledge and examples. This means that there is no existing ILP system, we know of, which directly uses pairs $(I, T_P(I))$ as input. For this reason it is necessary to transform these pairs into examples for ILP systems. This basically means to encode them as logical formulae.

For $I = \{A_1, A_2, \ldots, A_n\}$ and $T_P(I) = \{B_1, B_2, \ldots, B_m\}$ where $A_1, \ldots, A_n, B_1, \ldots B_m$ are atoms we get examples of the form $(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \rightarrow (B_1 \wedge B_2 \wedge \cdots \wedge B_m)$ or written in Horn clause form as $B_i \leftarrow (A_1 \wedge A_2 \wedge \cdots \wedge A_n)$ for $1 \leq i \leq m$. If the ILP systems support definite clauses as examples we can directly use this. Naturally, only finite interpretations I can be used, because only these can be written down.

We can generate "better" examples, i.e. facts instead of (possibly) non-unit clauses if we assume that T_P is the operator of a definite program. Starting with the empty interpretation we can now iteratively apply T_P. The elements of $T_P(I)$ we obtain this way are elements of the least Herbrand model and thus we can use them as positive examples. These are "better" examples in the sense that they make the learning task easier.

A problem with these approaches is that we get only positive examples. A quick idea could be to use $B \leftarrow (A_1 \wedge A_2 \wedge \cdots \wedge A_n)$ as negative examples for all $B \notin T_P(I)$. However this is not possible. A propositional logic program (this corresponds to the left upper box in Figure 4) suffices as illustration:

Example 3.9
Let P be the following program:

$$r \leftarrow q$$
$$q \leftarrow p$$

For $I = \{p\}$ we have $T_P(I) = \{q\}$, but nevertheless $r \leftarrow p$ is obviously not a negative example, because it follows from the program $(P \models r \leftarrow p)$. This means that using this as a negative example would render an ILP theory with $\Sigma = P$ (the best case we can expect) incorrect. □

We did not find an easy way to generate negative examples, so we will only learn from positive examples.

3.5 ILP Systems

In this section we will practically test several ILP systems with respect to the described test setup and draw conclusions from these tests. Please note that we restricted the research to ILP systems, which are freely available. There are too many different ILP systems to test them all, so we choose only five systems. The systems are chosen such that we get a good overview of the suitability of different ILP techniques for our task.

The ILP systems regarded here all use plain text files for specifying the background knowledge and the examples. Depending on the system, background knowledge, positive examples, and negative examples have to be written in different files. The systems are non-interactive, which means they do not generate questions for the user to obtain additional information. Due to the popularity of Prolog in logic programming most systems use a similar syntax.

Finding a correct theory naturally is a search process. Practical implementations of ILP theories often use various methods to narrow the search space. Usually this is done by letting the user specify additional information, which will be explained seperately for each ILP system we have tested.

3.5.1 CProgol

The first system we will investigate is CProgol. It was developed by Stephen Muggleton and is recognized as one of the best existing ILP systems. At the time of this writing it is still regularly updated. For the tests CProgol 4.4 was used. The following explanations are based on the CProgol manual [MF].

CProgol uses a Prolog-like syntax and has most Prolog predicates implemented in its core and available as background knowledge. Background knowledge and examples are all in one file. CProgol learns definite first order programs. Arbitrary Progol clauses are allowed as background knowledge. Positive examples are arbitrary definite examples and negative examples are represented as headleass Horn clauses, e.g. `:- father(stephen, peter)` means that `stephen` is not the father of `peter`.

As additional knowledge the user has to specify which predicates have to be learned and which predicates can be used in the definition of such predicates. Additionally the arity of all predicates and their argument types habe to be specified. Here is a simple example for specifiying in CProgol that you want to learn the predicate `father`:

Listing 3.10

```
modeh(1,father(+person,+person))?
```

The whole line is called a *mode declaration*. `modeh` means that the specified atom can occur as the head of a clause in the theory (also called *hypothesis*). The specification says that the predicate `father` is of arity two and both arguments are of *type* `person`. The arguments contain an additional sign, which can be either "+" indicating an input variable, "-" indicating an output variable or "#" indicating a constant value. By the above declaration we know that `father(X,Y)` can be used as the head of a clause. Every call to this should bind both arguments, i.e. a call to `father` should not contain

a variable. The `1` in the declaration above is the *recall number*. Is is used to limit the number of alternative solutions. In this case for a call to `father` the answer will be yes or no, so there is just one solution. If the user does not know a limit for the number of the alternative solutions `*` can be used instead of a number.

Besides head mode declarations there are also body mode declarations:

Listing 3.11
```
modeb(*,parent(-person,+person))?
modeb(*,parent(+person,-person))?
```

This says that the predicate `parent` can be used in the body of a hypothesis clause. If `parent(X,Y)` means that `X` is the parent of `Y`, then the first line says that this predicate can be used to find parents of a given child and the second line can be used to find children of a given parent.

Types are realized by writing down a predicate as background knowledge:

Listing 3.12
```
person(stephen).
person(peter).
person(barbara).
person(mark).
person(linda).
person(mary).
```

CProgol in new versions is able to learn from exclusively positive examples (called *positive only learning*), which is a quite common setting in general and especially suited for our task, because we have no easy way to generate negative examples. Positive only learning is turned on by adding this line to the specification:

Listing 3.13
```
:- set(posonly)?
```

Now we want to perform the first test with a propositional logic example. Let the following program P be given (see Figure 4 for an overview of the test setup):

Listing 3.14 (given program)
```
k :- s, p.
k :- f, g, e.
k :- z, p.
```

We use these randomly selected positive examples for CProgol:

Listing 3.15 (positive examples)
```
k :- s, p, e.
k :- s, p, g.
k :- s, p, z.
k :- s, p, e, g.
```

```
k :- z, p, e, s.
k :- z, p, s.
k :- z, p, f.
k :- z, p, e.
k :- f, g, e, s.
k :- f, g, e.
k :- f, g, e, z.
```

To repeat it: The examples are generated by using the T_P operator. The first example above exists, because $k \in T_P(\{s, p, e\})$. We obtain the following theory as result:

Listing 3.16 (result)

```
k :- p.
k :- e.
```

The theory says that whenever `p` or `e` is true, then `k` is also true. If we look at the examples we provided than we see that this is the case. This is because in the initial program every clause contains `p` or `e`, so every positive example must also contain `p` or `e`. However we immediately see that the induced theory is much too simple compared to the initial program. This means that the positive only learning does not give us the results we would like to have. It does not compensate the lack of negative examples good enough.

CProgol computes a rating for different clauses in each step and then takes the clause with the highest rating. The rating is primarily based on the information how many positive examples a clause explains and how many random instances (generated by the positive only learning algorithm) do not follow from the clause. Here are all clauses, which CProgol considers as the first clause of the theory together with some CProgol ratings:

Listing 3.17

```
188,136,1 k :- p.
188,128,1 k :- e.
187,124,1 k :- s.
185,104,1 k :- s, p.
182,76,1  k :- s, e.
181,72,1  k :- e, p.
181,72,1  k :- e, p.
175,56,1  k :- s, e, p.
0,192,191 k.
```

The first number is the overall rating (higher is better). The clause with the best rating is likely to be used in the final hypothesis. The second number is the rating for positive examples (higher is better) and the third number a rating for random instances (lower is better). One can see that the actually desired clauses get a slightly lower rating than the simple clauses, which are included in the final program. The third number shows

that there is a problem with the random instances for our test setup. It seems that CProgol does not generate good random instances of clauses in this case. We will come back to this topic later. CProgol offers the possibility to adjust parameters, but as this is a more fundamental problem there are no parameter adjustments, which will help in this case.

Next we will observe a first order example: the addition function.

Listing 3.18

```
add(X,0,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

Let us look at the mode declaration we need to specify:

Listing 3.19

```
:- modeh(1, add(+number, #number, -number))?
:- modeh(1, add(+number, s(+number), s(-number)))?
:- modeb(1, add(+number, +number, -number))?
```

The first line must be given, because of the first line in the definition of the add function. We explicitly allow a number as second argument. The other two arguments must be variables. In the second line we allow the use of the successor symbol `s` in the second and third argument of the function. The last line says that we can call `add` with three variables in the body of a hypotheses clause. Please note that these mode declarations already contain a lot of information. The advantage of these declarations is that they cut down the search space dramatically. This is a key to the success of CProgol and similar ILP systems. They allow a lot of domain specific knowledge to be used in these declarations. However the drawback is that in case you do not know these mode declarations (which is a common case) then you may not get the expected results. There has been research how to learn mode declarations, but CProgol does not yet support this feature.

Next we define the datatype `number`. This can easily be done by a recursive definition:

Listing 3.20

```
number(0).
number(s(X)) :- number(X).
```

Now we need examples as input for CProgol. We generate examples by using the emtpy interpretation and interpretations with one element as input for the T_P operator. We have $T_P(\emptyset) = \{(s^n(0), 0, s^n(0)) \mid n \in \mathcal{N}\}$, so we usually would get an infinite number of examples for each interpretation. We will write down only the first three examples for $n \leq 2$. For $I \neq \emptyset$ we always get a fourth example, because of the second clause in the definition of the `add` function. This gives us the following input file (lines starting with `%` are comments):

Listing 3.21

```
% empty interpretation
```

```
add(0,0,0).
add(s(0),0,s(0)).
add(s(s(0)),0,s(s(0))).

% {add(0,0,0)}
add(0,0,0) :- add(0,0,0).
add(s(0),0,s(0)) :- add(0,0,0).
add(s(s(0)),0,s(s(0))) :- add(0,0,0).
add(0,s(0),s(0)) :- add(0,0,0).

% {add(0,0,s(0))}
add(0,0,0) :- add(0,0,s(0)).
add(s(0),0,s(0)) :- add(0,0,s(0)).
add(s(s(0)),0,s(s(0))) :- add(0,0,s(0)).
add(0,s(0),s(s(0))) :- add(0,0,s(0)).

% {add(0,s(0),0)}
add(0,0,0) :- add(0,s(0),0).
add(s(0),0,s(0)) :- add(0,s(0),0).
add(s(s(0)),0,s(s(0))) :- add(0,s(0),0).
add(0,s(s(0)),s(0)) :- add(0,s(0),0).

% {add(0,s(0),s(0))}
add(0,0,0) :- add(0,s(0),s(0)).
add(s(0),0,s(0)) :- add(0,s(0),s(0)).
add(s(s(0)),0,s(s(0))) :- add(0,s(0),s(0)).
add(0,s(s(0)),s(s(0))) :- add(0,s(0),s(0)).

% {add(s(0),0,0)}
add(0,0,0) :- add(s(0),0,0).
add(s(0),0,s(0)) :- add(s(0),0,0).
add(s(s(0)),0,s(s(0))) :- add(s(0),0,0).
add(s(0),s(0),s(0)) :- add(s(0),0,0).

% {add(s(0),0,s(0))}
add(0,0,0) :- add(s(0),0,s(0)).
add(s(0),0,s(0)) :- add(s(0),0,s(0)).
add(s(s(0)),0,s(s(0))) :- add(s(0),0,s(0)).
add(s(0),s(0),s(s(0))) :- add(s(0),0,s(0)).

% {add(s(0),s(0),0)}
add(0,0,0) :- add(s(0),s(0),0).
add(s(0),0,s(0)) :- add(s(0),s(0),0).
add(s(s(0)),0,s(s(0))) :- add(s(0),s(0),0).
```

```
add(s(0),s(s(0)),s(0)) :- add(s(0),s(0),0).

% {add(s(0),s(0),s(0))}
add(0,0,0) :- add(s(0),s(0),s(0)).
add(s(0),0,s(0)) :- add(s(0),s(0),s(0)).
add(s(s(0)),0,s(s(0))) :- add(s(0),s(0),s(0)).
add(s(0),s(s(0)),s(s(0))) :- add(s(0),s(0),s(0)).
```

The examples look "odd", but this is the systematic way to generate them if we assume that the T_P operator is not an operator of a definite program, i.e. not monotonic. The theory we get is this one:

Listing 3.22

```
add(A,0,A).
add(A,s(B),s(C)) :- add(A,B,C).
```

This is exactly the definition of the desired `add` function. Now let us look at the examples if we assume that the T_P operator is monotonic (which is true in this case, because the add program is definite). Then we obtain examples by first computing $I = T_P(\emptyset)$, then compute $T_P(I)$) etc. ,i.e. we iteratively apply the T_P operator to resulting interpretations. We only do this three times and again only write down some of the (infinitely many) examples we get. The advantage is that this is exactly the standard setting for most of the ILP problems: the examples are ground facts. We use the following 12 examples:

Listing 3.23

```
add(0,0,0).
add(s(0),0,s(0)).
add(s(s(0)),0,s(s(0))).
add(s(s(s(0))),0,s(s(s(0)))).

add(0,s(0),s(0)).
add(s(0),s(0),s(s(0))).
add(s(s(0)),s(0),s(s(s(0)))).
add(s(s(s(0))),s(0),s(s(s(s(0))))).

add(0,s(s(0)),s(s(0))).
add(s(0),s(s(0)),s(s(s(0)))).
add(s(s(0)),s(s(0)),s(s(s(s(0))))).
add(s(s(s(0))),s(s(0)),s(s(s(s(s(0)))))).
```

If we run CProgol on these examples we also get the theory shown in Listing 3.22.

Listing 3.24

```
add(A,0,A).
add(A,s(B),s(C)) :- add(A,B,C).
```

An interesting question is raised by the observations made for CProgol. Why does it perform better for predicate logic than for propositional logic? From the observations made for the propositional logic example the author assumed that there is a problem with the random instances generated by CProgol. More concrete: CProgol uses the `modeh` declarations to generate possible ground atoms. For the propositional logic case the only possibility is `k`. For the declaration `:- modeh(1, add(+num, s(+num), s(-num)))?` CProgol will generate ground atoms as examples, where the second and third argument start with `s`. This assumption could be verified by reading [Mug01], where positive only learning is covered in detail. For the propositional logic case it would be better to not only generate atoms, but ground clauses, because otherwise the only random instance is the predicate itself. The approach CProlog uses is clearly directed towards learning first order theories. However by modifying the positive only learning algorithm to generate (non-unit) clauses instead of random instances for propositional logic, it could possibly also produce satisfying results for propositional logic.

3.5.2 Aleph

Aleph is written in Prolog. It works with separate text files for background knowledge, positive examples, and negative examples. It is the successor of PProgol, which itself is a Progol implementation of an earlier version of CProgol. For this reason its syntax is very similar to the one of Progol. Aleph has an active user base and is still regularly updated at the time of this writing. It has some notable differences with respect to our task compared to CProgol. Firstly in Aleph one can exactly specify for each predicate, which predicates can be used in its body. Secondly Aleph is able to learn mode declarations itself.

However there are also some serious restrictions. It is not directly possible to specify non-unit clauses as examples. Unfortunately this is exactly what we need if we assume that the network represents a non-monotonic T_P operator. Moreover learning mode declaration is only guaranteed to work if the background knowledge is ground and if positive only learning is not activated. In its current state Aleph does not provide an advantage over CProgol.

For the `add` example Aleph was not able to generate the desired theory with the examples, for which CProgol could induce the desired theory:

Listing 3.25

```
add(0,0,0).
add(s(0),0,s(0)).
add(s(s(0)),0,s(s(0))).
add(s(s(s(0))),0,s(s(s(0)))).

add(0,s(0),s(0)).
add(s(0),s(0),s(s(0))).
add(s(s(0)),s(0),s(s(s(0)))).
add(s(s(s(0))),s(0),s(s(s(s(0))))).
```

```
add(0,s(s(0)),s(s(0))).
add(s(0),s(s(0)),s(s(s(0)))).
add(s(s(0)),s(s(0)),s(s(s(s(0))))).
add(s(s(s(0))),s(s(0)),s(s(s(s(s(0)))))).
```

Instead it generated this theory:

Listing 3.26

```
add(A, 0, A).
add(0, s(0), s(0)).
add(A, s(B), s(A)) :- add(B, B, B).
add(0, s(s(0)), s(s(0))).
add(s(0), s(s(0)), s(s(s(0)))).
add(A, s(B), s(C)) :- add(B, A, C).
```

This shows that Aleph performs worse than CProgol on this concrete example. Adding more examples did not change the resulting theory.

3.5.3 FOIL

FOIL was written in 1993 by Quinlan and Cameron-Jones. (We tested version 6.) It is one of the most well-known and most succesful ILP systems. See [QCJ93] as a general reference.

FOIL learns from tuples. For one fixed predicate positive and negative examples are listed. Negative examples are optional. However FOIL does not use a probabilistic strategy to compensate negative examples, but the closed world assumption. This means all examples, which are not specified, are assumed to be negative. This in turn means that all possible examples should be specified, if negative examples cannot be provided, which is the case for our setup. For all data types every possible value must be specified in advance. Thus it is impossible to use infinite domains.

An example will illustrate this. Here is the (only) input file for FOIL when trying to learn the `member` predicate for lists:

Listing 3.27 (FOIL)

```
X: 1, 2, 3.
L: [111], [112], [113], [11], [121], [122], [123],
[12], [131], [132], [133], [13], [1], [211],
[212], [213], [21], [221], [222], [223], [22], [231],
[232], [233], [23], [2], [311], [312], [313], [31],
[321], [322], [323], [32], [331], [332], [333], [33],
[3], *[].

member(X,L)
1,[1]
3,[3]
```

```
...
3,[331]
3,[333]
;
1,[]
1,[3]
...
1,[323]
1,[332]
.

...
```

(*Instead of ... there is additional code.*)

In the example one first specifies the possible values for `X` and `L`. `X` is an element of a list and `L` is a list with these elements. Elements marked with `*` can be part of the induced theory. Further down in the input file one can find the header `member(X,L)` (standing for `X` is an element of `L`). Below this header all positive examples are listed until the line with the semicolon. From there on, until the line with the single dot, the negative examples follow.

It is not clear how to use FOIL for rule extraction, because the input syntax does not allow the specification of non-unit clauses as examples. It can only be used in the case we have facts as examples (i.e. we assume T_P is an operator of a definite program). However in this case there is still the restriction that only a finite domain can be used and all positive examples need to be given. So for most cases we cannot use it to extract rules.

3.5.4 FFOIL

FFOIL was written by Ross Quinlan in 1996 (we tested version 2). It is specialized on the learning of functions, i.e. relations where all values except the last one are input values. FFOIL is written to only learn from positive examples. Other than that it is very similar to FOIL. We could not run any succesful task relevant tests with FFOIL, so we cannot recommend it for our setup.

3.5.5 Golem

Golem is a classical ILP system. It was first described by Muggleton and Feng [MF92]. It uses simple mode declarations, but uses completely different ILP techniques like CProgol. Golem requires the background knowledge to be ground. Unfortunately Golem does not support positive only learning. By using the `add` example it was tested, if Golem is nevertheless able to induce good theories, but this was not the case, i.e. it was not able to extract general rules from the examples.

3.6 Conclusions

As a summary we now list some of the positive and negative observations we made when using ILP systems for extracting a logic program from a neural network representing a T_P operator.

Positive Results/Observations:

- The fact that the task could be at least partly solved using ILP techniques is itself a success. Since this was (probably) not done before we could not expect very good results.
- CProgol is able to learn definite first order programs from clauses as positive examples.

Negative Results/Observations:

- The observed ILP systems only learn definite programs (if at all). This is a restriction, because it requires the T_P operator represented by the network to be monotonic. However using learning algorithms for the neural network may change it into a nonmonotonic network, so we cannot guarantee this assumption in general.
- For ILP systems it is not important if the operator of the extracted program is equal to the operator, which is represented by the network. This is naturally the case, because the ILP systems we have observed are concerned with first order logic semantics. Maintaining the T_P operator allows the use of different semantics for the resulting program. However this is only a problem if the resulting program is not definite.
- For the special case of propositional logic none of the programs provided good results, although this case is obviously simpler than first order case. We noted that is probably possible to modify CProgol such that it produces better results for this case. The next part of this document will investigate extraction methods for propositional logic in detail. Our hope behind this research is that there are good extraction algorithms with provable results.
- Most ILP systems do not support positive only learning with non-unit ground clauses as examples.
- Only finite interpretations can be used to generate examples.
- From the infinitely many examples we may get for each pair $(I, T_P(I))$ we still have to intelligently collect good examples, which is itself not an easy task.
- CProgol found a solution for the `add` program, but for this we need to encode a lot of knowledge in the mode declarations. Such knowledge may not always exist. The question, if ILP systems, which do not need such knowledge, still give the same good results, is still open.

As a conclusion we can say that ILP is an option for extracting definite first order logic programs. CProgol currently seems to be the best (tested) program for this task.

4 Exact Extraction of Programs

Section 3 has shown that ILP techniques give us a partial success for extracting definite first order programs. However we feel the need of researching the extraction task (for the special case that the network represents a T_P operator) in a more theoretical way. We are interested in extraction algorithms, which return correct and minimal programs. Correctness in this context means that the operator of the network is the same like the operator of the program. Minimality means that the program should have minimal size (which we will later define precisely). Currently these questions are not even solved for propositional logic, so we will only focus on this base case. Of course one of the hopes is that similar algorithms can be used for the first order case.

4.1 Preliminaries

Since this section talks only about propositional logic programs, we explicitly spell out the definition of an immediate consequence operator for propositional logic.

Definition 4.1 (T_P operator for propositional logic programs)
T_P is a mapping from interpretations to interpretations defined in the following way for an interpretation I and a propositional logic program P:

$$\begin{aligned} T_P(I) :=& \{q \mid q \leftarrow p_1, \ldots, p_n, \neg r_1, \ldots, \neg r_m \in P, n \geq 0, m \geq 0, \\ & \{p_1, \ldots, p_n\} \subseteq I, \{r_1, \ldots, r_n\} \cap I = \emptyset\} \end{aligned}$$

□

For convenience the corresponding definition for the special case of definite programs is given:

Definition 4.2 (T_P operator for definite propositional logic programs)
T_P is a mapping from interpretations to interpretations defined in the following way for an interpretation I and a definite propositional logic program P:

$$\begin{aligned} T_P(I) :=& \{q \mid q \leftarrow p_1, \ldots, p_n \in P, n \geq 0, \\ & \{p_1, \ldots, p_n\} \subseteq I\} \end{aligned}$$

□

An important property of T_P for definite programs P is montonicity, i.e. $I \subseteq J$ implies $T_P(I) \subseteq T_P(J)$.

Analogously to the T_P operator we will also write down the notion of subsumption for the case of propositional logic. This definition is easier than the first order definition, because we do not need substitutions.

Definition 4.3 (subsumption for propositional logic)
Assume that we have a Horn clause C_1: $h \leftarrow p_1, \ldots, p_a, \neg q_1, \ldots, \neg q_b$ and a Horn clause C_2: $h \leftarrow r_1, \ldots, r_c, \neg s_1, \ldots, \neg s_d$ with $\{p_1, \ldots, p_a\} \subseteq \{r_1, \ldots, r_c\}$ and $\{q_1, \ldots, q_b\} \subseteq \{s_1, \ldots, s_d\}$. Then C_1 *subsumes* C_2. □

If it is clear from the context we will sometimes say "program" or "logic program" instead of "propositional logic program".

4.2 Reduced Programs

Assume we have the following programs with only a single predicate p and a single Horn clause:

$$P_1 = \{p \leftarrow p\}$$

$$P_2 = \{p \leftarrow p, p\}$$

There are only two possible interpretations for a single predicate, namely $I_1 = \emptyset$ and $I_2 = \{p\}$. The T_P operator for both programs is the same, because $T_{P_1}(I_1) = T_{P_2}(I_1) = \emptyset$ and $T_{P_1}(I_2) = T_{P_2}(I_2) = \{p\}$. Our conclusion is, that in the general case one T_P operator represents several (in fact infinitely many) different programs.

In Section 4.3 we will show that all definite programs having an operator T_P can be reduced to exactly one program. For this we introduce the notion of a reduced program and show how to construct it.

Definition 4.4 (reduced propositional logic program)
A reduced program P is a program with two additional properties:

1. There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in P, such that C_1 subsumes C_2.
2. A predicate symbol does not appear more than once in a body of a clause. □

One can construct a reduced program Q from every program P by using the following algorithm:

Algorithm 4.5 (constructing a reduced program)
For an arbitrary propositional logic program P perform the following reduction steps as long as possible:

1. If there are clauses C_1 and C_2 with $C_1 \neq C_2$ in P and C_1 subsumes C_2, then remove C_2.
2. If a literal appears twice in the body of a clause, then remove it once.
3. If a predicate and its negation appear in the body of a clause, then remove this clause. □

Obviously the resulting program is reduced. If a program P was reduced to a program Q by this algorithm we say "Q is a reduced program of P". Please note that from now on when we speak about (possible) clauses of an extracted program we will often implicitly assume that these do not contain a literal twice in their body. With this simple restriction the number of possible clauses is finite in propositional logic.

Proposition 4.6 (correctness of reduction)
If Q is a reduced propositional logic program of P then $T_P = T_Q$.

PROOF For proving the proposition we will show that each reduction step in Algorithm 4.5 does not influence the immediate consequence operator. More formally we call P_1 the program before a reduction step and P_2 the program after a reduction step. We must show $T_{P_1} = T_{P_2}$ for each of the four possible steps. For this we have to show $T_{P_1}(I) = T_{P_2}(I)$ for an arbitrary interpretation I.

1. Let C_1: $h \leftarrow p_1, \ldots, p_a, \neg q_1, \ldots, \neg q_b$ and C_2: $h \leftarrow r_1, \ldots, r_c, \neg s_1, \ldots, \neg s_d$ with $\{p_1, \ldots, p_a\} \subseteq \{r_1, \ldots, r_c\}$ and $\{q_1, \ldots, q_b\} \subseteq \{s_1, \ldots, s_d\}$. Because C_2 is removed, we know $P_2 = P_1 \backslash \{C_2\}$.

   $T_{P_2}(I) \subseteq T_{P_1}(I)$: This is equivalent to $q \in T_{P_2}(I) \rightarrow q \in T_{P_1}(I)$. Because P_1 has one clause more than P_2 and is otherwise identical this is obvious. (Adding clauses cannot delete elements in $T_{P_2}(I)$).

   $T_{P_1}(I) \subseteq T_{P_2}(I)$: This is equivalent to $q \in T_{P_1}(I) \rightarrow q \in T_{P_2}(I)$. We assume $q \in T_{P_1}(I)$ and want to show $q \in T_{P_2}(I)$. $q \neq h$ implies $q \in T_{P_2}(I)$, because for Horn clauses with heads other than h nothing has changed by removing C_2. So we can assume $q = h$. Further we know that $h \in T_{P_1 \backslash \{C_1;C_2\}}(I)$ implies $h \in T_{P_2}(I)$ by the same argument like in the other direction of the proof. This means we can assume $h \notin T_{P_1 \backslash \{C_1;C_2\}}(I)$. Because of this and $h \in T_{P_1}(I)$ we have $h \in T_{\{C_1;C_2\}}(I)$. This means that ($\{p_1, \ldots, p_a\} \subseteq I$ and $\{q_1, \ldots, q_b\} \cap I = \emptyset$) or ($\{r_1, \ldots, r_c\} \subseteq I$ and $\{s_1, \ldots, s_d\} \cap I = \emptyset$). In both cases we have $\{p_1, \ldots, p_a\} \subseteq I$ and $\{q_1, \ldots, q_b\} \cap I = \emptyset$, so we have $h \in T_{\{C_1\}}(I)$. Hence $h \in T_{P_2}(I)$.

2. Here $T_{P_1} = T_{P_2}$ is trivial, because sets are used in the definition of T_P, so it does not matter if an element exists twice or once.

3. Let C be a clause of the form $h \leftarrow p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_m$ with $p \in \{p_1, \ldots, p_n\}$ and $p \in \{q_1, \ldots, q_m\}$. No interpretation fulfills $p \in I$ and $p \notin I$ as required by the definition of T_P, so the clause can be safely removed. ∎

Why do we need the notion of a reduced program? We have seen that reducing programs maintains the T_P operator. We later want to extract an unknown program from a T_P operator, but instead of extracting an arbitrary program we will try to extract a reduced program. Simply speaking reduced programs are easier to read and smaller, because they contain less superfluos clauses respectively literals in the body of clauses. $p \leftarrow p$ and $p \leftarrow p, p$ are syntactically different, but we prefer the first over the latter. Similarly we do not want to have $q \leftarrow p_1$ and $q \leftarrow p_1, p_2$ in one program, because the latter clause does not have any influence on the T_P operator of the program if the first clause is already present. In this sense $q \leftarrow p_1, p_2$ is superfluous. Logic programs written by humans are usually reduced.

4.3 Extracting Definite Propositional Logic Programs

What we want to show in this section is, that a given T_P operator for an unknown program P corresponds to exactly one reduced definite program. We can divide this work in two steps. At first we will constructively show, that there is a definite program for each monotonic operator. Later we will prove that the program we get is indeed the only definite program which has the given operator.

We start by giving an extraction algorithm and later prove properties of it.

Algorithm 4.7 (extracting a reduced definite program from T_P)
Let T_P be an operator of an unknown definite propositional logic program P B_P be the set of all predicates in P and Q be an empty set.

For all interpretations $I \subseteq B_P$ ordered by $|I|$ starting with $|I| = 0$ do the following:

Let $I = \{p_1, \ldots, p_n\}$. For every $q \in T_P(I)$ check, if a clause $q \leftarrow q_1, \ldots, q_m$ with $\{q_1, \ldots, q_m\} \subseteq I$ is in Q. If this is not the case, then add the clause $q \leftarrow p_1, \ldots, p_n$ to Q. □

Obviously this algorithm terminates, because there are only finitely many interpretations. Please note that we require the input of the algorithm to be an operator of a definite program. (More precisely for proving the two following propositions we just require the input of the algorithm to be a monotonic mapping $M : 2^{B_P} \rightarrow 2^{B_P}$.) Also note that the actual order of the interpretations is not important as long as an interpretation is always treated before any interpretation having more elements.

There are interesting points regarding the efficiency of the presented algorithm. The author recommends to order the predicates (the order can be arbitrary). This makes it possible to write a method, which returns the successor of an interpretation I with respect to $|I|$ and predicate order. Thus it is not necessary to store all possible interpretations, hence the space complexity is very low (basically only Q and the current interpretation need to be stored). The bottleneck is the time complexity, which is exponential with respect to the number of predicates. However it is also exponential with respect to the maximum length of clauses in Q. (This is because for an input $|I| = n$ the algorithm only generates Horn clauses of length n.) Thus if we know a limit n of the number of elements in a body of a Horn clause in advance, we can reduce time complexity and maintain the properties, we will prove, by stopping the algorithm if $|I| > n$.

Proposition 4.8 (correctness of the extraction algorithm)
Let T_P be the input of Algorithm 4.7 and the program Q its ouput, then $T_P = T_Q$.

PROOF We have to show $T_P(I) = T_Q(I)$ for an arbitrary interpretation $I = \{p_1, \ldots, p_n\}$.

$T_P(I) \subseteq T_Q(I)$: This is equal to $q \in T_P(I) \rightarrow q \in T_Q(I)$. Assume we have a predicate q in $T_P(I)$. We know that the algorithm will treat I and q (because for every interpretation I every element in $T_P(I)$ is investigated). Then we have to distinguish to cases.

Case 1: There already exists a clause $q \leftarrow q_1, \ldots, q_m$ with $\{q_1, \ldots, q_m\} \subseteq I$ in Q. Then by definition $q \in T_Q(I)$.

Case 2: There is no such clause. Then the clause $q \leftarrow p_1, \ldots, p_n$ added to Q, so we have again $q \in T_Q(I)$.

$T_Q(I) \subseteq T_P(I)$: Analogously to the first part of the proof we now have a predicate q in $T_Q(I)$ and want to show it is in $T_P(I)$. If $q \in T_Q(I)$ we have by definition of T_Q a clause $q \leftarrow q_1, \ldots, q_m$ with $\{q_1, \ldots, q_m\} \subseteq I$. This means that the extraction algorithm must have treated the case $q \in T_P(J)$ with $J = \{q_1, \ldots, q_m\}$. Because T_P is monotonic (it is the operator of a definite program) and $J \subseteq I$ we have $T_P(J) \subseteq T_P(I)$, so q is also an element of $T_P(I)$. ■

Proposition 4.9
The output of Algorithm 4.7 is a reduced definite propositional logic program.

PROOF Obviously the output of the algorithm is a definite program, because it generates only definite Horn clauses. We have to show that the resulting program is reduced. By contradicition we assume Q is not reduced. According to Definition 4.4 there are two possible reasons for this:

Case 1: A predicate symbol appears more than once in the body of a Horn clause. This is impossible, because the algorithm does not generate such clauses (sets do not contain elements twice).

Case 2: There are two different Horn clauses C_1 and C_2 in Q, such that C_1 subsumes C_2. Let C_1: $h \leftarrow p_1, \ldots, p_a$ and C_2: $h \leftarrow q_1, \ldots, q_b$ with $\{p_1, \ldots, p_a\} \subseteq \{q_1, \ldots, q_b\}$. As abbreviations we use $I = \{p_1, \ldots, p_a\}$ and $J = \{q_1, \ldots, q_b\}$. Because of case 1 we know $|I| = a$ and $|J| = b$ (all elements in the body of a Horn clause are different). Thus we have $|I| < |J|$, because C_1 and C_2 are not equal. This means the algorithm has treated I (and $h \in T_P(I)$) before J (and $h \in T_P(J)$). C_1 was generated by treating I and h, because C_1 exists and can only be generated through I and h (otherwise the body respectively head of the Horn clause would be different). Later the case J and h was

treated. The algorithm checks for clauses $h \leftarrow r_1, \ldots, r_m$ with $\{r_1, \ldots, r_m\} \subseteq J$. C_1 is such a clause, because $I \subseteq J$, so C_2 is not added to Q. Because (by the same argument as above) C_2 can only be generated through J and h, C_2 cannot be a clause in Q, which is a contradiction and completes the proof. ∎

Proposition 4.8 and 4.9 have shown, that the output of the extraction algorithm is in fact a reduced definite program, which has the desired operator. While the algorithm itself is an important practical result of this paper, there is an interesting theoretical result, which we will prove next.

Proposition 4.10
For any operator T_P of a definite propositional logic program P there is exactly one reduced definite propositional logic program Q with $T_P = T_Q$.

PROOF Assume we have an operator T_P of a definite program P. With Algorithm 4.7 and the Propositions 4.8 and 4.9 it follows that there is a reduced definite program Q with $T_P = T_Q$. We have to show that there cannot be more than one program with this property.

To prove this we assume (by contradiction) that we have two different reduced definite programs P_1 and P_2 with $T_P = T_{P_1} = T_{P_2}$. Two programs being different means that there is at least one Horn clause existing in one of the programs, which does not exist in the other program. Without loss of generality we assume that there is a clause C_1 in P_1, which is not in P_2. C_1 is an arbitrary definite Horn clause of the form $h \leftarrow p_1, \ldots, p_m$. By definition of T_P we have $h \in T_{P_1}(\{p_1, \ldots, p_m\})$. Because T_{P_1} and T_{P_2} are equal we also have $h \in T_{P_2}(\{p_1, \ldots, p_m\})$. This means that there is a clause C_2 of the form $h \leftarrow q_1, \ldots, q_n$ with $\{q_1, \ldots, q_n\} \subseteq \{p_1, \ldots, p_n\}$ in P_2. Applying the definition of T_P again this means that $h \in T_{P_2}(\{q_1, \ldots, q_n\})$ and $h \in T_{P_1}(\{q_1, \ldots, q_n\})$. Thus we know that there must be a clause C_3 of the form $h \leftarrow r_1, \ldots, r_o$ with $\{r_1, \ldots, r_o\} \subseteq \{q_1, \ldots, q_n\}$ in P_1.

C_3 subsumes C_1, because it has the same head and $\{r_1, \ldots, r_o\} \subseteq \{q_1, \ldots, q_n\} \subseteq \{p_1, \ldots, p_m\}$. We know that by our assumption C_1 is not equal to C_2, because C_1 is not equal to any clause in P_2. Additionally we know that $|\{p_1, \ldots, q_m\}| = m$ and $|\{q_1, \ldots, q_n\}| = n$, because P_1 and P_2 are reduced, i.e. no predicate appears more than once in a body of a Horn clause. So we have $\{q_1, \ldots, q_n\} \subset \{p_1, \ldots, p_m\}$. Because C_3 has at most as many elements in its body as C_2, we know that C_1 is not equal to C_3. That means that P_1 contains two different clauses C_1 and C_3, where C_3 is a reduced Horn clause of C_1. This is a contradiction to P_1 being reduced. ∎

This shows that each algorithm extracting reduced definite programs from T_P must return the same result like Algorithm 4.7. From this proposition another result follows easily, showing that the reduction we defined is optimal for definite programs with respect to T_P and a straightforward size measure of programs.

Definition 4.11
The size of a program P is the sum of the number of all literals in all Horn clauses in P. A program P is (strictly) smaller than a program Q, if its size is (strictly) smaller than the size of Q. A program is *minimal*, if there is no strictly smaller program with the same immediate consequence operator. □

Corollary 4.12
If P is a reduced definite propositional logic program, then it is minimal.

PROOF By contradiction we assume there exists a program Q, which has operator T_P and P is not smaller than Q. By Algorithm 4.5 and Proposition 4.6 we know that Q can be reduced. The resulting program Q_{red} is definite, smaller (this was not proved, but is obvious) and has the operator T_P. From Proposition 4.10 we know that there is only one reduced definite program with operator T_P, so we have $P = Q_{red}$. Because Q_{red} is smaller than Q, P is also smaller than Q. ■

4.4 Extracting Normal Propositional Logic Programs

In this section we will try to extend the extraction to normal propositional logic programs.

Remark 4.13
It is easy to see that Proposition 4.10 does not hold for normal programs. In general there can be more than one reduced propositional logic program Q with $T_P = T_Q$. These are two reduced programs having the same T_P operator:

$$P_1 = \{p\}$$

$$P_2 = \{p \leftarrow p; p \leftarrow \neg p\}$$

□

This means we should investigate other extraction methods. We will see that the extraction is indeed a lot harder for normal programs compared to definite programs. The first result we need is that for any given mapping we can construct a program with this operator.

Proposition 4.14
Let B_P be a finite set of predicates. For every mapping $M : 2^{B_P} \to 2^{B_P}$ from Herbrand interpretations to Herbrand interpretations we can construct a propositional logic program P with $T_P = M$.

PROOF The construction of P works as follows: P is initialized as an empty set. For every interpretation I with $I = \{r_1, \ldots, r_a\}$ and $B_P \setminus I = \{s_1, \ldots, s_b\}$ and every element p in $M(I)$ we add a clause $p \leftarrow r_1, \ldots, r_a, \neg s_b, \ldots, \neg s_m$ to P.

For this program we have to show $T_P(J) = M(J)$ for an arbitrary interpretation J with $J = \{t_1, \ldots, t_c\}$ and $B_P \setminus J = \{u_1, \ldots, u_d\}$.

$M(J) \subseteq T_P(J)$ Assume $q \in M(J)$. We want to show $q \in T_P(J)$. Since $q \in M(J)$ means (by the above construction) that P contains a clause $q \leftarrow t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ we obtain $q \in T_P(J)$.

$T_P(J) \subseteq M(J)$ Assume $q \in T_P(J)$. We want to show $q \in M(J)$. $q \in T_P(J)$ means that there is a clause $q \leftarrow v_1, \ldots, v_e, \neg w_1, \ldots, \neg w_f$ with $\{v_1, \ldots, v_e\} \subseteq \{t_1, \ldots, t_c\}$ and $\{w_1, \ldots, w_f\} \cap I = \emptyset$, i.e. $\{w_1, \ldots, w_f\} \subseteq \{u_1, \ldots, u_d\}$. Since every clause in P has all elements of B_P in its body the only possible clause, which fulfills these requirements is $q \leftarrow t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$. This clause was added to P by the above construction, because there is an interpretation $\{t_1, \ldots, t_c\} = J$ such that $q \in M(J)$. ■

A short example will illustrate this simple construction:

Example 4.15
Let $B_P = \{p, q\}$ and the mapping M be defined by the following:

$$\begin{aligned} M(\emptyset) &= \{p\} \\ M(\{p\}) &= \{q\} \\ M(\{q\}) &= \emptyset \\ M(\{p, q\}) &= \{p, q\} \end{aligned}$$

We obtain the following program P:

$$\begin{aligned} p &\leftarrow \neg p, \neg q \\ p &\leftarrow p, q \\ q &\leftarrow p, \neg q \\ q &\leftarrow p, q \end{aligned}$$

It is now easy to verify $T_P = M$ in this example. □

4.4.1 Reduction Methods

The construction above is already a correct extraction method as we have proved. However we are interested in getting small programs with the same operator. For this we define, as in the case of definite programs, reductions on programs. Remark 4.13 has shown that Definition 4.4 (reduced propositional logic programs) is not strong enough: There exist reduced programs, which can obviously be further reduced. This means we need stronger requirements.

Definition 4.16 (α-reduced programs)
An α-reduced program P is program with the following properties:

1. There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in P where C_1 is of the form $p \leftarrow q, r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ and C_2 is of the form $p \leftarrow \neg q, t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ with $\{r_1, \ldots, r_a\} \subseteq \{t_1, \ldots, t_c\}$ and $\{s_1, \ldots, s_b\} \subseteq \{u_1, \ldots, u_d\}$.
2. There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in P where C_1 is of the form $p \leftarrow \neg q, r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ and C_2 is of the form $p \leftarrow q, t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ with $\{r_1, \ldots, r_a\} \subseteq \{t_1, \ldots, t_c\}$ and $\{s_1, \ldots, s_b\} \subseteq \{u_1, \ldots, u_d\}$.
3. There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in P, such that C_1 subsumes C_2.
4. No predicate symbol appears more than once in the body of a clause in P. □

In this definition the third and fourth point are the same like for reduced programs. The first and second point are new. These points can be used if there are two clauses where one contains an atom p and the second one its negation. If this is the case and additionally the body without p is a subset of the body of the other clause without $\neg p$ (or vice versa) then we know that the program can be further reduced. An example will illustrate this.

Example 4.17
The most simple example is the one of Remark 4.13:

$$\begin{aligned} P_1 &= \{p\} \\ P_2 &= \{p \leftarrow p; \\ &\qquad p \leftarrow \neg p\} \end{aligned}$$

Here P_2 is not α-reduced. Let C_1 be $p \leftarrow p$ and C_2 be $p \leftarrow \neg p$. Then the first condition of Definition 4.16 is (trivially) not fulfilled, because $\emptyset \subseteq \emptyset$. P_2 should be reduced to P_1. Let us consider another example:

$$\begin{aligned} P_1 &= \{p \leftarrow \neg p, \neg r; \\ &\qquad p \leftarrow \neg q, \neg r\} \\ P_2 &= \{p \leftarrow \neg p, \neg r; \\ &\qquad p \leftarrow p, \neg q, \neg r\} \end{aligned}$$

Again P_2 is not α-reduced. Let C_1 be $p \leftarrow p, \neg q, \neg r$ and C_2 be $p \leftarrow \neg p, \neg r$. Then we have $\{\neg r\} \subseteq \{\neg r, \neg q\}$ for the bodies of C_1 and C_2 without p respectively $\neg p$. P_1 is α-reduced and has the same operator. □

For completeness we now give the algorithm for constructing an α-reduced program.

Algorithm 4.18 (constructing an α-reduced program)
For an arbitrary propositional logic program P perform the following reduction steps as long as possible:

1. If there are two clauses C_1 and C_2 such that point 1 of Definition 4.16 is fulfilled, then remove $\neg q$ in the body of C_2.
2. If there are two clauses C_1 and C_2 such that point 2 of Definition 4.16 is fulfilled, then remove q in the body of C_2.
3. If there are clauses C_1 and C_2 with $C_1 \neq C_2$ in P and C_1 subsumes C_2, then remove C_2.
4. If a literal appears twice in the body of a clause, then remove it once.
5. If a predicate and its negation appear in the body of a clause, then remove this Horn clause. □

If a program P was reduced to a program Q by this algorithm we say "Q is an α-reduced program of P".

Proposition 4.19 (correctness of reduction)
If Q is an α-reduced propositional logic program of P then $T_P = T_Q$.

Proof Only the first two points have to be proved. For the other points see the proof of Proposition 4.6. We only show the first point, because the second point is completely analogous. Let $C_1 \in P$ be $p \leftarrow q, r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ and $C_2 \in P$ be $p \leftarrow \neg q, t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ with $\{r_1, \ldots, r_a\} \subseteq \{t_1, \ldots, t_c\}$ and $\{s_1, \ldots, s_b\} \subseteq \{u_1, \ldots, u_d\}$ as written in the definition of α-reduced programs. Let C_2' be the clause $p \leftarrow t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$. We have to show that replacing C_2 by C_2' in P does not change the operator. Because we have already shown point 3 of the reduction algorithm we know, that we can add a the clause C_3 of the form $p \leftarrow q, t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ to P without changing the T_P operator, because C_1 subsumes C_3. This means we have $T_P = T_{P \cup \{C_3\}}$. But then the only difference between C_2 and C_3 is that C_2 contains $\neg q$ in its body and C_3 contains q in its body. Now $T_{\{C_2;C_3\}}(I) = T_{\{C_2'\}}(I)$ is easy to see. By replacing C_2 and C_3 by C_2' it follows $(T_P =) T_{P \cup \{C_3\}} = T_{(P \setminus C_2) \cup C_2'}$, because in a program we can replace two clauses by an equivalent clause with respect to the immediate consequence operator without changing the immediate consequence operator of the whole program. Thus we have shown that we can apply the first reduction rule without changing the immediate consequence operator (by temporarily adding C_3). ■

An interesting observation is that adding a clause C to an α-reduced program and then executing all possible reductions for C and other clauses in the program does not necessarily give an α-reduced program.

Example 4.20
Let C be $p \leftarrow p, \neg q, \neg r$ and P be the following program:

$$p \leftarrow \neg p, \neg q, \neg r$$
$$p \leftarrow p, \neg q, r$$
$$p \leftarrow p, q, \neg r$$

Reducing C and the first clause we get the new clause $C' = p \leftarrow \neg q, \neg r$. C' can be reduced with the second clause and the result is $C'' = p \leftarrow p, \neg q$. C'' cannot be reduced with any other clause. This is the program we have obtained so far:

$$p \leftarrow \neg q, \neg r$$
$$p \leftarrow p, \neg q$$
$$p \leftarrow p, q, \neg r$$

However this program is not α-reduced, because clause 3 of this program can be further reduced (with clause 2). □

Now a straightforward way for defining an extraction algorithm is to build up clauses according to the construction in Proposition 4.14 and then apply the reduction algorithm (for reducing space complexity it is also possible to add only some clauses, then reduce, then add more clauses, reduce etc.). This obviously yields a correct algorithm for obtaining an α-reduced program by Proposition 4.14 and 4.19. The question is, if we get the same properties like for the case of definite programs, i.e. a minimal program. The following example shows that this is not always the case.

Example 4.21
The following programs have the same immediate consequence operator:

P_1:	P_2:	P_3:
$p \leftarrow \neg p, \neg r$	$p \leftarrow \neg p, \neg r$	$p \leftarrow \neg p, \neg r$
$p \leftarrow p, r$	$p \leftarrow p, r$	$p \leftarrow p, r$
$p \leftarrow \neg p, q$	$p \leftarrow q, r$	$p \leftarrow q, r$
		$p \leftarrow \neg p, q$

All of these programs are α-reduced. However we see that an α-reduced program is not always minimal, because P_3 is clearly larger than P_2. Another observation is that P_3 can be easily reduced to P_2 by removing a clause. However this cannot be done by α-reduction. Therefore we will introduce an even stronger reduction than α-reduction.□

Definition 4.22 (β-reduced programs)
A β-reduced program P is a program with the following properties:

1. There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in P where C_1 is of the form $p \leftarrow q, r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ and C_2 is of the form $p \leftarrow \neg q, t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ with $\{r_1, \ldots, r_a\} \subseteq \{t_1, \ldots, t_c\}$ and $\{s_1, \ldots, s_b\} \subseteq \{u_1, \ldots, u_d\}$.

2. There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in P where C_1 is of the form $p \leftarrow \neg q, r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ and C_2 is of the form $p \leftarrow q, t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ with $\{r_1, \ldots, r_a\} \subseteq \{t_1, \ldots, t_c\}$ and $\{s_1, \ldots, s_b\} \subseteq \{u_1, \ldots, u_d\}$.

3. There is no clause $C \in P$ with $T_{P\setminus\{C\}} = T_P$.

4. No predicate symbol appears more than once in the body of a clause in P. □

Compared to α-reduced programs only the third point has changed. It is now strictly stronger by saying that not only clauses, which are subsumed by another clause in the program are deleted, but every clause, which does not change the overall operator of the program. The points 1,2 and 4 are like before.

Algorithm 4.18 can be changed for β-reduction, such that we get a correct algorithm. The only new reduction step is the removal of clause C, if point 3 of the above definition is true, but this trivially does not change the immediate consequence operator of the program. We can check point 3 by removing any clause C in the program and then test if $T_P = T_{P\setminus\{C\}}$ (which is of course decidable in propositional logic).

Is it now the case that we always get minimal programs? Unfortunately the answer is no, as an example will show.

Example 4.23
P_1 and P_2 have the same immediate consequence operator:

P_1:	P_2:
$p \leftarrow \neg p, r$	$p \leftarrow \neg p, r$
$p \leftarrow q, \neg r$	$p \leftarrow q, \neg r$
$p \leftarrow p, \neg q$	$p \leftarrow p, \neg r$
	$p \leftarrow \neg q, r$

The two programs above are β-reduced, but have different size. In this example we see that the third and fourth clause of P_2 can be replaced by the third clause of P_1 in this context. This leads to another reduction method. □

Definition 4.24 (γ-reduced programs)
A γ-reduced program P is a program with the following properties:

1. There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in P where C_1 is of the form $p \leftarrow q, r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ and C_2 is of the form $p \leftarrow \neg q, t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ with $\{r_1, \ldots, r_a\} \subseteq \{t_1, \ldots, t_c\}$ and $\{s_1, \ldots, s_b\} \subseteq \{u_1, \ldots, u_d\}$.

2. There are no clauses C_1 and C_2 with $C_1 \neq C_2$ in P where C_1 is of the form $p \leftarrow \neg q, r_1, \ldots, r_a, \neg s_1, \ldots, \neg s_b$ and C_2 is of the form $p \leftarrow q, t_1, \ldots, t_c, \neg u_1, \ldots, \neg u_d$ with $\{r_1, \ldots, r_a\} \subseteq \{t_1, \ldots, t_c\}$ and $\{s_1, \ldots, s_b\} \subseteq \{u_1, \ldots, u_d\}$.

3. There is no clause $C \in P$ with $T_{P\setminus\{C\}} = T_P$.

4. There are no clauses $C_1 \in P$ and $C_2 \in P$ such that there exists a clause C_3 with $T_{C_1;C_2} = T_{C_3}$.

5. No predicate symbol appears more than once in the body of a clause in P. □

Only one new point was added compared to β-reduction.

Again one can find a correct reduction algorithm for γ-reduction. An easy implementation can work this way: Remove two arbitrary clauses C_1 and C_2 (all combinations have to be tested!) from the current program and add an arbitrary clause C_3 (all possible clauses have to be tested!). If this does not change the consequence operator then replace C_1 and C_2 by C_3. (Of course C_3 should not contain a predicate symbol more than once in its body otherwise there are infinitely many possibilites.)

Again we are interested wether γ-reduction always gives always a minimal program? Unfortunately the answer is no again. Although it becomes increasingly hard to find counterexamples they do exist:

Example 4.25
P_1 and P_2 have the same immediate consequence operator:

P_1:	P_2:
$p \leftarrow q, r, \neg s$	$p \leftarrow \neg p, q, r$
$p \leftarrow \neg q, \neg r$	$p \leftarrow \neg p, \neg r, s$
$p \leftarrow \neg p, q, s$	$p \leftarrow p, \neg q$
$p \leftarrow p, \neg q, s$	$p \leftarrow p, r, \neg s$
	$p \leftarrow \neg q, \neg r$

The two programs above are γ-reduced, but have different size. Note that clauses 1, 2 and 4 in P_2 can be replaced by clauses 1 and 3 of P_1. We could now go on and include the case that three clauses can be replaced by two clauses in a definition of δ-reduction, but we will stop now. Probably we would still be able to find a counterexample and additionally γ-reduction is already computationally extremely expensive. □

What we have done so far is to define three possible types of reduction. There is a tradeoff, which has to be made between the power of reduction with respect to minimality and the computing power necessary to compute the reduction. The good thing is that different kinds of reduction can be combined. So it is of course possible to first use α-reduction until a program is α-reduced, then use β-reduction until the program is β-reduced and finally use γ-reduction.

As a final result for the reduction methods presented, we can give a straightforward extraction algorithm for normal programs:

Algorithm 4.26 (extracting a reduced program from T_P)
Let T_P be an operator of an unknown propositional logic program P, B_P be the set of all predicates in P and Q be an empty set.

Construct the program Q as defined in Proposition 4.14 and then perform a reduction method on Q (which can be either α-, β- or γ-reduction). □

Depending on wether α-, β- or γ-reduction is chosen one obtains a different algorithm, but as explained above these can and should be combined. Please note that we did not formally define β and γ reduction, but gave the intuition in the explanations of the corresponding definitions.

4.4.2 Pruning Possible Clause Bodies

As a base for further algorithms we look at a strategy of making the search for a minimal program with a given operator more efficient. For this we will restrict the set of clause bodies we consider for each predicate.

Definition 4.27 (allowed clause)
Let T_P be an immediate consequence operator, $B = p_1, \ldots, p_a, \neg q_1, \ldots, \neg q_b$ be a clause body and h be a predicate. We call B *allowed with respect to* h *and* T_P if it has the following property:

For every interpretation I with $\{p_1, \ldots, p_a\} \subseteq I$ and $\{q_1, \ldots, q_b\} \cap I = \emptyset$ we have $h \in T_P(I)$ and there is no allowed clause body $B' = r_1, \ldots, r_c, \neg t_1, \ldots, \neg t_d$ for h with $B' \neq B$ such that $\{r_1, \ldots, r_c\} \subseteq \{p_1, \ldots, t_a\}$ and $\{t_1, \ldots, t_d\} \subseteq \{q_1, \ldots, q_b\}$. □

The idea is that we do not want to allow clauses, which clearly lead to an incorrect T_P operator and we do not want to allow clauses, for which a shorter allowed clause exists. The intuition is of course that any minimal program can only consist of allowed clauses. The following example will illustrate this construction:

Example 4.28 (pruning clause bodies)
We will use the operator of the programs in Example 4.21 and compute the allowed clause bodies with respect to this operator and the predicate p. This is the operator:

$$\begin{aligned} T_P(\emptyset) &= \{p\} \\ T_P(\{p\}) &= \emptyset \\ T_P(\{q\}) &= \{p\} \\ T_P(\{r\}) &= \emptyset \\ T_P(\{p,q\}) &= \emptyset \\ T_P(\{p,r\}) &= \{p\} \\ T_P(\{q,r\}) &= \{p\} \\ T_P(\{p,q,r\}) &= \{p\} \end{aligned}$$

Now we want to reduce the number of possible clause bodies for clauses with head p. Intuitively we do not want to consider clause bodies, which are "wrong". For example the clause body p, q is "wrong", because adding the clause $p \leftarrow p, q$ would lead to $p \in T_P(\{p, q\})$, which is not correct. Furthermore we also do not want to consider clause bodies for which shorter ones can be used, because we want to construct a minimal program. For instance the clause body p, q, r should not be used, because p, r can be used as well and is shorter. We need to write down all possible clause bodies (of course not considering those, which contain a predicate more than once) and check, which of them can be pruned:

clause body	evaluation
$\square$	False, $p \notin T_P(\{p\})$.
p	False, because $p \notin T_P(\{p\})$.
q	False, because $p \notin T_P(\{p, q\})$.
r	False, because $p \notin T_P(\{r\})$.
$\neg p$	False, because $p \notin T_P(\{r\})$.
$\neg q$	False, because $p \notin T_P(\{p\})$.
$\neg r$	False, because $p \notin T_P(\{p\})$.
p, q	False, because $p \notin T_P(\{p, q\})$.
p, r	OK.
q, r	OK.
$p, \neg q$	False, because $p \notin T_P(\{p\})$.
$p, \neg r$	False, because $p \notin T_P(\{p\})$.
$q, \neg p$	OK.
$q, \neg r$	False, because $p \notin T_P(\{p, q\})$.
$r, \neg p$	False, because $p \notin T_P(\{r\})$.
$r, \neg q$	False, because $p \notin T_P(\{r\})$.
$\neg p, \neg q$	False, because $p \notin T_P\{r\})$.
$\neg p, \neg r$	OK.
$\neg q, \neg r$	False, because $p \notin T_P(\{p\})$.
p, q, r	Not considered, because p, r is smaller.
$p, q, \neg r$	False, because $p \notin T_P(\{p, q\})$.
$p, \neg q, r$	Not considered, because p, r is smaller.
$\neg p, q, r$	Not considered, because q, r is smaller.
$p, \neg q, \neg r$	False, because $p \notin T_P(\{p\})$.
$\neg p, q, \neg r$	Not considered, because $\neg p, q$ is smaller.
$\neg p, \neg q, r$	False, because $p \notin T_P(\{r\})$.
$\neg p, \neg q, \neg r$	Not considered, because $\neg p, \neg r$ is smaller.

The left side of the table shows all possible clause bodies and the right side lists if this clause body is allowed. If this is not the case the reason is given. We see that the number of possible clause bodies is reduced from 27 to 4 in this case. □

4.4.3 A Greedy Algorithm

So far we have only considered algorithms, which build up a program P from T_P in several steps by reducing a large correct program (top-down approach). Another possible approach is to build up a program clause by clause to get closer to the desired operator.

The first thing we can do is to handle each predicate seperately. More precisely for an arbitrary predicate p we create a subprogram, which only consists of clauses with head p. Joining all subprograms gives us the overall program. This construction can be done, because of the definition of the immediate consequence operator. For a given predicate h, an immediate consequence operator T_P and an interpretation I we know that $h \in T_P(I)$ only depends on clauses with head h. The following definition introduces an operator, where the resulting interpretation is restricted to one predicate:

Definition 4.29
Let $q \in B_P$ be a predicate and T_P be an immediate consequence operator. We define a function $T_P^q : 2^{B_P} \to 2^{B_P}$:

$$T_P^q(I) = \begin{cases} \emptyset & \text{if } q \notin T_P(I) \\ \{q\} & \text{if } q \in T_P(I) \end{cases}$$ □

Let us see how we use this definition: Let $B_P = \{q_1, \ldots, q_m\}$ be the set of all predicates and T_P a given operator. Now we try to find *subprograms* Q_i for each predicate q_i such that $T_{Q_i} = T_P^{q_i}$. If we construct the union $Q = Q_1 \cup \cdots \cup Q_m$ of such programs we obtain a program with operator T_P. This is because for an arbitrary predicate r we have the following arguments: $r \in T_P(I)$ if and only if $r \in T_P^r(I)$ if and only if $r \in T_{Q_r}$.

For the bottom-up approach of creating a program we define a score function. This function takes as arguments a clause and the program we have constructed so far and returns a value. This definition only makes sense if we use it for allowed clauses. For an allowed clause there is a set of interpretations, which does not yet behave like the desired operator with respect to a certain predicate and the program we have constructed so far, but would behave correctly if we add the clause to the constructed program. The more such interpretations exist the higher the score should be. The intuitive meaning will later become clearer in Example 4.32 in this section.

Definition 4.30 (score)
Let B_P be a set of predicates. The *score* of a clause with respect to a program Q is defined as follows:

$$score(h \leftarrow p_1, \ldots, p_m, \neg q_1, \ldots, \neg q_n; Q) = \\ |\{I \mid I \subseteq 2^{B_P} \text{ and } \{p_1, \ldots, p_n\} \subseteq I \text{ and } \{q_1, \ldots, q_n\} \cap I = \emptyset \text{ and } h \notin T_Q(I)\}|$$ □

We can now give a greedy algorithm. It creates subprograms for each predicate by iteratively adding the clauses with the highest score. If there are several clauses with the highest score we prefer those with less literals, because we want to construct a small program.

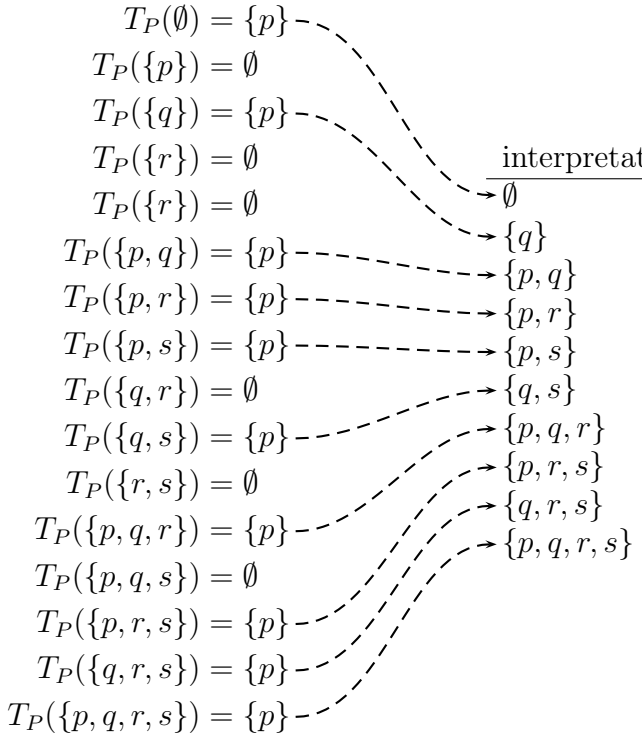
Algorithm 4.31 (Greedy Extraction Algorithm)
Let T_P and $B_P = \{q_1, \ldots, q_m\}$ be the input of the algorithm.

Initialize: $Q = \emptyset$
Foreach predicate $q_i \in B_P$:
    construct the set S_i of allowed clause bodies for q_i
    initialize: $Q_i = \emptyset$
    repeat:
        Determine a clause C of the form $h \leftarrow B$ with $B \in S_i$ with the highest score with respect to Q_i. If several clauses have the highest score, then choose one with the smallest number of literals.
        $Q_i = Q_i \cup \{C\}$
    until $T_{Q_i} = T_P^{q_i}$
    $Q = Q \cup Q_i$ □

Example 4.32
Let the operator shown below on the left side be the input of the extraction algorithm. From this operator we can compute the set S of allowed clause bodies:

$$S = \{p, r; \neg p, \neg r, \neg s; q, \neg p, \neg r; q, \neg r, \neg s; p, q, \neg s; p, s, \neg q; q, s, \neg p; q, r, s\}$$

The two tables below the operator show two different runs of the algorithm. In each step the score for the allowed clauses, which are not already in the constructed program, is given. (The score of the clause which is added to the constructed program Q is in boldface.) As an example the score for $p, q, \neg s$ in step I of the first run is 2, because $p \in T_P(p, q)$ and $p \in T_P(p, q, r)$. It goes down to 1 in the second step, because we have $Q = \{p, r\}$ and therefore $p \in T_Q(p, q, r)$ at this point. Intuitively this means we would only gain one additional interpretation by adding $p \leftarrow p, q, \neg s$. The table on the right summarizes the interpretations gained by the two different runs in each step.



interpretation	run 1	run 2
$\emptyset$	III	II
$\{q\}$	II	II
$\{p, q\}$	IV	IV
$\{p, r\}$	I	I
$\{p, s\}$	V	V
$\{q, s\}$	II	III
$\{p, q, r\}$	I	I
$\{p, r, s\}$	I	I
$\{q, r, s\}$	VI	III
$\{p, q, r, s\}$	I	I

clause body	I	II	III	IV	V	VI
p, r	**4**					
$\neg p, \neg r, \neg s$	2	2	**1**			
$q, \neg p, \neg r$	2	**2**				
$q, \neg r, \neg s$	2	2	1	**1**		
$p, q, \neg s$	2	1	1	1	0	0
$p, s, \neg q$	2	1	1	1	**1**	
$q, s, \neg p$	2	2	1	1	1	**1**
q, r, s	2	1	1	1	1	1

clause body	I	II	III	IV	V
p, r	**4**				
$\neg p, \neg r, \neg s$	2	**2**			
$q, \neg p, \neg r$	2	2	1	0	0
$q, \neg r, \neg s$	2	2	1	**1**	
$p, q, \neg s$	2	1	1	1	0
$p, s, \neg q$	2	1	1	1	**1**
$q, s, \neg p$	2	2	**2**		
q, r, s	2	1	1	0	0

The example is constructed such that there are two different runs of the algorithm, which return programs with different size for the same operator. The first run produced a program with six clauses and 17 literals. The second run produces a program with five clauses and 14 literals. This shows that this algorithm does not always return a minimal program, which is an expected result, because the algorithm is greedy, i.e. it always chooses the clause, which seems to be the best according to the score heuristic, without looking at the effects of this decision. We also see that the algorithm is not deterministic, because there may be several clauses with the highest score and the lowest number of literals (e.g. in step III of run 1). □

As a sidenote the algorithm can easily be extended to construct programs, which approximate the correct operator. As a stopping criterion for creating a subprogram the algorithm above requires $T_P^{q_i} = T_{Q_i}$. Instead of this strict criterion one could stop if all the remaining clauses only have a small score (e.g. the maximum score is 1) or if $T_P^{q_i}$ and T_{Q_i} are equal for a certain percentage (e.g. 95%) of interpretations. As the initial motivation is to extract knowledge from neural networks it may be desirable to extract rules, which do not overfit, but are shorter and hence more understandable.

4.4.4 An Intelligent Program Search Algorithm

So far we could not guarantee minimality of the resulting program for any of the extraction algorithms. One approach to realize this is to use a program search algorithm. This means we search all possible programs with increasing size and stop if we have found a program, which has the desired operator. This seems to be inefficient, but there are two points, which motivate the specification of such an algorithm:

1. The previous algorithms did not guarantee that a minimal program is extracted. An important goal of this paper is to present such an algorithm as it was done for definite propositional logic programs. If the (probably) most efficient way to do this is a program search algorithm, then it should be specified.

2. There are ways to make program search more efficient.

There are three things which will be done to make the search more efficiency:

1. We will only search programs with clauses, which do not contain predicates more than once in their body. We can do this, because a minimal program never contains such clauses. (see Algorithm 4.5 and Proposition 4.6)

2. We create subprograms for every predicate seperately. This is exactly the same approach like in the greedy algorithm. Of course if all such subprograms are minimal then the resulting program is also minimal. Why does this make program search much more efficient? The reason is that the complexity now depends on the size of the minimal subprogram of the most complex predicate instead of the size of the whole minimal program.

3. Again we only consider allowed clause bodies when constructing programs. We can do this, because we have seen every minimal program can only consist of clauses with allowed clause bodies.

Now we will give the program search algorithm. It essentially uses the techniques of the greedy algorithm, but instead of using a heuristic (the score function) to add clauses to subprograms it performs a full program search.

Algorithm 4.33 (program search algorithm)
Let T_P and $B_P = \{q_1, \ldots, q_m\}$ be the input of the algorithm.

Initialize: $Q = \emptyset$
Foreach predicate $q_i \in B_P$:
construct the set S_i of allowed clause bodies for q_i
initialize: $n_i = 0$
repeat:
For every set $(a_1, \ldots, a_n)$ with $\Sigma_{j=1}^{n} j a_j = n$:
Construct programs Q_i which have a_j clauses with j literals $(0 \leq j \leq n)$, where each clause has head q_i and its body is an element of S_i, until a program with $T_P^{q_i} = T_{Q_i}$ is found or all possible programs were constructed.
increment n_i
until there is a program Q_i with $T_P^{q_i} = T_{Q_i}$
$Q = Q \cup Q_i$ □

The sum $\Sigma_{j=1}^{n} j a_j = n$ in the algorithm is only for iterating through all possibilities, how a program with n literals can be build. For instance if $n = 6$ there can be one clause with six literals, two clauses with three literals, one clause with four literals and another with two literals etc. Of course only those a_i $(1 \leq i \leq n)$ can have a value greater than 0, for which clauses with allowed bodies of length $i-1$ exist ("-1", because of the head).

Proposition 4.34
Let T_P be the input for Algorithm 4.33. Then its ouput Q is a minimal propositional logic program with $T_P = T_Q$.

PROOF This directly follows from the construction of the algorithm and the explanations given above. ■

The next example describes very briefly how the algorithm works.

Example 4.35
Assume the T_P operator of Example 4.28 is given as input to the algorithm:

$$
\begin{aligned}
T_P(\emptyset) &= \{p\} \\
T_P(\{p\}) &= \emptyset \\
T_P(\{q\}) &= \{p\} \\
T_P(\{r\}) &= \emptyset \\
T_P(\{p,q\}) &= \emptyset \\
T_P(\{p,r\}) &= \{p\} \\
T_P(\{q,r\}) &= \{p\} \\
T_P(\{p,q,r\}) &= \{p\}
\end{aligned}
$$

For the predicates q and r there are no allowed clauses, so the algorithm would construct empty subprograms for those predicates. For the predicate p we computed the allowed clauses: $S = \{p, r; q, r; q, \neg p; \neg p, \neg r\}$. The algorithm now tries to build a subprogram for p with an increasing number n of literals. For $n = 0$ this obviously fails. If 3 is not a divider of n this fails, too, because every element of S has exactly two elements, so only clauses with 3 literals are possible. For $n = 3$ the algorithm tests every single clause with bodies in S. For $n = 6$ the algorithm tests every combination of two clauses with bodies in S. All these combiations do not give the desired operator T_P^p as one can manually verify. For $n = 9$ the algorithm will be succesful, because we have seen that there are programs with this size and operator T_P^p in Example 4.21. □

Again the algorithm can easily be extended to create programs, which approximate the correct operator. Instead of the stopping criterion $T_P^{q_i} = T_{Q_i}$ one can require that the operators are equal only for a certain percentage of interpretations.

4.4.5 A Result Regarding Minimal Programs

The previous algorithms have shown that extracting normal programs is a lot harder than extracting definite programs. The next proposition gives an intuition why this is the case. We will show that there is in general not a unique minimal program. As a counterexample we use the same operator like in the Examples 4.21, 4.28 and 4.35.

Proposition 4.36
There can be more than one minimal propositional logic program with a given immediate consequence operator.

PROOF The claim is that for the programs P_1 and P_2 in Example 4.21 there is no smaller program, i.e. a program with less literals, with the same immediate consequence operator. To prove this we have to show that there is no smaller program having the operator $T_{P_1} (= T_{P_2})$. (We will write T_P instead of T_{P_1} or T_{P_2} from now on.) In Example

4.28 we have computed the set S of allowed clause bodies for the predicate p (it is obvious that a minimal program with this operator consists only of clauses with head p): $S = \{p, r; q, r; q, \neg p; \neg p, \neg r\}$. In particular there is no clause body with one or three literals. This means for a program to be smaller than P_1 or P_2, which have the size of nine literals, it must have one or two clauses where each clause has exactly three literals (exactly two literals in the body).

We further see that a program must have the clause $p \leftarrow \neg p, \neg r$, because it is the only one for which we get $T_P(\emptyset) = \{p\}$. We also need the clause $p \leftarrow p, r$, because it is the only one which gives us $T_P(\{p, r\}) = \{p\}$. But for the program $Q = \{p \leftarrow \neg p, \neg r; p \leftarrow p, r\}$ we have $p \notin T_Q(\{q, r\})$. This means that there is no program with two or less clauses with operator T_P, which completes the proof. ∎

4.5 Conclusions

We have given algorithms for creating a program from operators of definite and normal programs. The extraction algorithm for definite programs (Algorithm 4.7) had a lot of nice properties: It returns a correct and reduced program. Moreover there is always exactly one reduced program with minimal size for each operator.

As usual in logic programming most of the good properties for definite programs are lost for normal programs. We could prove that there is in general not only one program with minimal size for a given operator. We evaluated several different reduction strategies, which work similar like the easier reduction for definite programs. However while these approaches work, they cannot guarantee that we get a minimal program. To make this possible we gave a program search algorithm (Algorithm 4.33), which has this property, but is slow if there are predicates, which need complex subprograms. For the case where a minimality guarantee is not needed we gave a greedy algorithm (Algorithm 4.31), which builds up a program by always adding the best possible clause according to the score heuristic. The greedy algorithm as well as the program search algorithm can also extract programs, which approximate the correct operator, if this is desired.

Even with their negative properties normal programs are usually more desirable in the context of network extraction, because we have shown that there exist normal programs for arbitrary mappings from (finite) interpretations to (finite) interpretations, which means we can extract a program from every network.

The complexity of the given algorithms is not always easy to estimate. All the algorithms need the whole consequence operator as input, so they are exponential with respect to the number $|B_P|$ of predicates, because there exist $2^{|B_P|}$ interpretations. The algorithm for extracting definite programs only performs a computationally inexpensive check for each such interpretation. For extracting normal programs the complexity is higher. The algorithm working with reductions (Algorithm 4.26) needs to do a lot more work for each interpretation depending on the type of reduction used (α-, β- or γ-reduction). For β- and γ-reduction the complexity depends strongly on the number of possible clauses, because it is necessary to check wether one (respectively two) clauses can be replaced by an arbitrary possible clause. The complexity of the pruning technique presented in Section 4.4.2 also depends on the number of possible clauses (and is

therefore exponential with respect to the number of predicates), because for each possible clause we have to decide wether it is allowed or not. This decision itself involves searching through the whole operator in the worst case, so the worst case complexity for computing all allowed clauses is $2^{|B_P|} \times 2^{|B_P|}$. The greedy algorithm and the program search algorithm roughly depend on the number of allowed clauses and the minimal size of the largest subprogram. The author wants to emphasize that better approximations of the complexity and suitability of the presented algorithms will need further more detailed investigations (ideally combined with practical tests).

How do the algorithms in this section relate to the ILP techniques used in Section 3? Of course the similarity is that both have the same goal to extract a logic program from a network representing an immediate consequence operator. However there are some important differences. The first one is that ILP techniques are suited for handling first order logic while the algorithms in this section can only handle propositional logic. Another difference is that for the ILP techniques we usually only gave some pairs $(I, T_P(I))$ for generating examples, while the algorithms in this section usually consider the whole consequence operator as given. The reason for this is that the algorithms in this section are correct with respect to the operator of the generated program, i.e. the given operator and the operator of the generated program are equal. The programs induced by ILP techniques do not have this property, but instead are correct (as defined in Definition 3.3) with respect to a given set of examples.

References

[Apt97] K.R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[BH04] S. Bader and P. Hitzler. Logic programs, iterated function systems, and recurrent radial basis function networks. *Journal of Applied Logic*, 2(3):273–300, 2004.

[BHH04] S. Bader, P. Hitzler, and S. Hölldobler. The integration of connectionism and first-order knowledge representation and reasoning as a challenge for artificial intelligence. In L. Li and K.K. Yen, editors, *Proceedings of the Third International Conference on Information*, pages 22–33, Tokyo, Japan, November/December 2004. International Information Institute.

[dGBG01] A.S. d'Avila Garcez, K. Broda, and D.M. Gabbay. Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125:155–207, 2001.

[HK94] S. Hölldobler and Y. Kalinke. Towards a new massively parallel computational model for logic programming. In *Proceedings of the ECAI94 Workshop on Combining Symbolic and Connectionist Processing*, pages 68–77. ECCAI, 1994.

[HKS99] S. Hölldobler, Y. Kalinke, and H.-P. Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11:45–58, 1999.

[Mah88] M. J. Maher. Equivalences of logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, CA, 1988.

[MB88] S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In *ML88*, pages 339–351. MK, 1988.

[MF] S. Muggleton and J. Firth. *CProgol 4.4: a tutorial introduction*. Department of Computer Science, University of York, United Kingdom.

[MF92] S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *ILP*, pages 281–298. AP, 1992.

[MR94] S.H. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[Mug01] S.H. Muggleton. Learning from positive data. *Machine Learning*, 2001. Accepted subject to revision.

[NCdW97] S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Springer, 1997.

[QCJ93] J.R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Machine Learning: ECML-93, European Conference on Machine Learning, Proceedings*, volume 667, pages 3–20. Springer-Verlag, 1993.

[Qui92] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1992.

[RN03] S. Russel and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[Roj02] R. Rojas. *Neural networks: a systematic introduction*. Springer, 2002.

[Sha91] E.Y. Shapiro. Inductive inference of theories from facts. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 199–255. MIT, 1991.

Statement of Academic Honesty

Hereby I declare that this is my work and that I did not use any other sources than the onces cited.