Logic Programming

Jürgen Giesl

Summer Semester 2022

Research Group Computer Science 2

RWTH Aachen University

Contents

Chapter 1

Introduction

For computer scientists, the knowledge of different families of programming languages is necessary for several reasons:

- The familiarity with different concepts of programming languages allows them to better express their own ideas in software development.
- The background knowledge of different programming languages is necessary to choose the most suitable language in a concrete project.
- Having learned some conceptually different programming languages makes it easy and fast to learn new ones in the future.
- One task of computer scientists is to design new programming languages. This is only possible on the basis of the existing languages.

In general, one distinguishes between *imperative* and *declarative* programming languages (where declarative languages are further divided into functional and logic languages). In imperative languages, the programs consist of sequences of instructions that are executed after each other and change the values of the variables in memory. Most of the programming languages used today are based on this principle which has a direct connection to the classical computer architecture tracing back to John von Neumann.

In declarative programming, the programs consist of a specification of *what* should be computed. *How* the computation works is determined by the interpreter or compiler, respectively. Declarative programming languages are therefore problem oriented instead of machine oriented.

On the one hand, all programming languages are equally expressive, which means that every program can be written in any of the languages typically used. On the other hand, the languages vary in their suitability for different areas of application. For example, imperative languages such as C are used for fast machine-oriented programming, since the programmer can (and must) directly take over the memory management. In other programming languages, this task is performed automatically (by the compiler or the runtime system). This allows faster program development, which is less error-prone. However, the resulting programs are usually also less efficient (i.e., they need more runtime and memory).

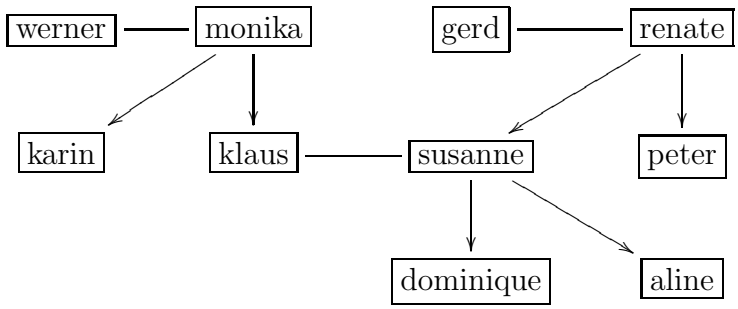
While in functional programming a program implements a function (and also in imperative programming a program implicitly corresponds to a function, which changes the values of variables), logic programs consist of rules for defining relations (i.e., a logic program is a collection of statements about the relation between different objects). During the execution of the program, these statements are used to answer and solve queries.

We will consider the language Prolog as an example for logic programming. It is mainly used in artificial intelligence (e.g., for expert systems, language processing, deductive data bases, etc.).

In the following, the basic principles of Prolog are briefly explained to motivate the content and structure of the lecture. Several implementations of Prolog are available. One free implementation that we recommend is SWI Prolog, available at

`https://www.swi-prolog.org`.

The principle of logic programming is that the programmer only describes the logic relations of a problem to be solved, i.e., the *knowledge base*. This does not require any knowledge of machine-related details of the computer. As an example, we consider the following knowledge base about family relations.



The arrows go from the mothers to their children and horizontal lines indicate married persons. For simplicity, we assume that there are no divorces, no children of unmarried couples, and no same-sex marriages.

Facts and Queries

To represent this knowledge base, the language of *predicate logic* is used in Prolog. In fact "Prolog" stands for "Programming in Logic". So the knowledge base consists of logic formulas (so-called *clauses*). More precisely, there are two kinds of clauses in programs: *facts* (which make statements about objects) and *rules* (which allow to infer new facts from known facts). In our example, the knowledge base (initially) consists only of facts:

```
female(monika).
female(karin).
female(renate).
female(susanne).
female(aline).
```

```
male(werner).
male(klaus).
male(gerd).
male(peter).
male(dominique).

married(werner, monika).
married(gerd, renate).
married(klaus, susanne).

motherOf(monika, karin).
motherOf(monika, klaus).
motherOf(renate, susanne).
motherOf(renate, peter).
motherOf(susanne, aline).
motherOf(susanne, dominique).
```

A fact always starts with the name of the property or relation (the so-called *predicate symbol*) . Directly after that (without spaces) the objects are mentioned which have this property. At the end, there is a dot. The next fact is in the next line (or at least there is a space to separate it from the previous clause). In Prolog, objects (like `monika`) and properties (like `motherOf`) start with lower case letters. The number of arguments is called the *arity* of the predicate symbol. A predicate corresponds to a function whose result is "true" or "false". Note that the relations are directed. Thus, if an Object 1 is in a relation to Object 2, then this is not necessarily the case the other way around. For example, we have `motherOf(monika,karin)` but not `motherOf(karin,monika)`. Comments in Prolog start with a `%` and go to the end of the line, or they are enclosed in `/*` and `*/`.

As mentioned, running a logic program means that the user poses *queries* to the program. So it is a "dialogue-oriented" programming language. A possible query would be for example: "Is `gerd male`?". In Prolog queries start with "`?-`". This symbol is followed by the statements (formulas) which should be proved. Here, one can prove several formulas at once by separating the formulas by commas. At the end of the query there is a dot. In Prolog you would therefore ask the query

```
?- male(gerd).
```

Prolog generates the answer by performing a logic proof based on the existing knowledge (i.e., based on the problem description given by the programmer). Thus, the task of the computer is to find the solution, i.e., the computer is the *inference machine* and "computing" means "proving" in Prolog. For these proofs, Prolog uses the techniques of "*unification*" and "*resolution*". In the query above, the computer would answer with `true`. But for the query

```
?- married(gerd, monika).
```

one will get the answer `false`. The computer assumes that its knowledge base contains all relevant knowledge about the world. So if a certain statement is not *entailed* by its

knowledge, it is considered *false*. For that reason, the statement `married(gerd, monika)` is regarded as being false.

In order to be able to ask queries to a program, one has to load the program first. If the program is located in a file named "`file.pl`", one has to enter "`consult(file).`" (or "`[file].`" for short) in the Prolog interpreter or compiler. Afterwards the knowledge from this knowledge base is available. The file must start with a lower case letter.

Variables in Programs

The knowledge base can also contain *variables*. Variables in Prolog start with capital letters or an underscore. As an example, we add the following fact to the above knowledge base:

```
human(X).
```

Variables in the knowledge base stand for *all* possible assignments, i.e., this fact means "All objects are humans". If one now asks the query

```
?- human(gerd).
```

then one gets the answer `true` as expected. But the query "`?- human(5).`" would also lead to the result `true`.

Same variables in the same facts mean equality. So the fact "`likes(X,Y).`" means "Everyone likes everyone". On the other hand, "`likes(X,X).`" means "Everyone likes themselves." But same variables in different clauses have nothing to do with each other.

Variables in Queries

It is also possible to let the program compute solutions itself. To this end, one uses variables in queries. As an example, we consider the query

```
?- motherOf(X, susanne).
```

This corresponds to the question "Who is the mother of `susanne`?" or, respectively, "Is there an assignment of the variable `X` such that `X` is the mother of `susanne`?". Now the computer does not (only) answer with `true`, but it searches for such an assignment of `X`. The answer is therefore `X = renate`. Variables in the knowledge base are thus *universally quantified* and variables in queries are *existentially quantified*.

As another example, we consider the query

```
?- motherOf(renate, Y).
```

that is, the question "Who are the children of Renate?". The computer now answers with "`Y = susanne`".

But this is not the only possible solution. If one wants to compute further solutions, one must enter a semicolon. The computer now looks for further solutions and one obtains "`Y = peter`". Thus, `motherOf` is not a function where the role of input and output are fixed, but it is a relation. Here it is not fixed what input and output are, but this depends on the query. So one can use the same program to compute all children of a woman and to compute the mother of a child. Of course one can also ask "`?- motherOf(X,Y).`". So it is

up to the user of the program to decide which values he or she wants to ask for and which values he wants the program to compute.

To process such queries, Prolog searches the knowledge base from top to bottom and returns the first answer that it finds. *So the clauses of the knowledge base are processed from top to bottom.*

As a further example, we consider the query

```
?- human(X).
```

Any possible instantiation of the variable `X` would be a solution now. Here, the computer computes the "most general" solution `true`. The reason is that the statement is true for every instantiation of `X`. The program always tries to find the *most general* answers to the queries.

Combination of Questions

As already mentioned, one can also prove several statements at the same time (i.e., one can combine questions). An example is the following query:

```
?- married(gerd,F), motherOf(F,susanne).
```

So the question is whether there is a woman `F` who is married to `gerd` and is also the mother of `susanne`. The variable `F` must be instantiated in the same way in the whole query. The procedure of Prolog is to solve the leftmost part "`married(gerd,F)`" of the query first. In this way, a solution for `F` is found. With this solution one then tries to solve "`motherOf(F,susanne)`". If this does not succeed, one performs a *backtracking* step and tries to find another solution for `F`, which also satisfies "`married(gerd,F)`". *So statements in a query are processed from left to right.*

As another example we consider the following question, i.e., who is the grandmother of `aline`:

```
?- motherOf(Grandma,Mom), motherOf(Mom,aline).
```

Here we first try `Grandma = monika, Mom = karin`. Then we backtrack until the solution `Grandma = renate, Mom = susanne` is found. If the two questions `motherOf(Grandma,Mom)` and `motherOf(Mom,aline)` had been swapped, the solution would have been found faster (without backtracking).

Rules

Besides *facts*, the knowledge base can also contain *rules*. Rules serve to deduce new knowledge from existing knowledge. As an example we consider the father-child relationship. This relationship could of course be defined specifically for all objects, but it is much shorter and clearer to formulate it by a general rule. (In particular, this would not be possible in any other way with an infinite number of objects or with a set of objects that is growing or shrinking later.) The following rule states "A person `V` is father of a child `K`, if he is married to a woman `F`, and this woman is the mother of the child `K`."

```
fatherOf(V,K) :- married(V,F), motherOf(F,K).
```

The symbol ":-" thus means "if" and rules formulate "if - then" relations. If the conditions on the right-hand side of the rule are true, then the statement on the left-hand side is true as well. The left-hand side is called the *head* of the rule and the right-hand side is its *body*. The conditions in the body are separated by commas and the rule ends with a dot. The meaning of a rule `p :- q,r.` is: If `q` and `r` hold, then `p` holds as well.

During the execution of programs in Prolog (i.e., for performing proofs), rules are applied backwards. To show that the left-hand side of a rule holds, it must be shown that the right-hand side holds as well (*backward chaining*). As an example, let us consider the query

```
?- fatherOf(gerd,susanne).
```

To prove this statement, due to the rule for `fatherOf`, one must find an `F` such that the statements `married(gerd,F), motherOf(F,susanne)` are true. This corresponds exactly to the query `?- married(gerd,F), motherOf(F,susanne).` Prolog therefore answers with "`true`". Analogously, the query

```
?- fatherOf(gerd,Y).
```

would result in the answers `Y = susanne` and `Y = peter`.

Several Rules for the Same Predicate

So far we have seen a rule where the head holds if the *conjunction* of the conditions holds. Now we consider the case where the head follows from the *disjunction* of two conditions. For this we use several rules for the same predicate symbol. As an example, we consider a predicate `parent`, where `parent(X,Y)` is true if `X` is the mother or father of `Y`. The rules for this are as follows:

```
parent(X,Y) :- motherOf(X,Y).
parent(X,Y) :- fatherOf(X,Y).
```

If we now ask the query

```
?- parent(X, susanne).
```

we get the answers `X = renate` and `X = gerd`. Note that the order of the clauses also influences the search and the order of the solutions.

Recursive Rules

Recursion is an important programming technique in Prolog. As an example, we define a predicate `ancestor`. Here `V` is an ancestor of `X` if `V` is a parent of `X` or if there is a `Y` such that `V` is a parent of `Y` (i.e., `V` has a child `Y`) and `Y` is an ancestor of `X`. The translation of this rule into Prolog results in:

```
ancestor(V,X) :- parent(V,X).
ancestor(V,X) :- parent(V,Y), ancestor(Y,X).
```

The query

```
?- ancestor(X, aline).
```

now means "Who are the ancestors of `aline`?". Here, Prolog finds the following answers:

```
X = susanne;
X = klaus;
X = monika;
X = renate;
X = werner;
X = gerd
```

Characteristics of Logic Programs

To summarize, logic programs have the following properties:

1. (Pure) logic programs have no control structures to manage the control flow of programs. The programs are only collections of facts and rules, which are processed from top to bottom (respectively, from left to right).

2. Logic programming has evolved from automatic theorem proving and when a logic program is executed it tries to prove a query. During this proof, solutions for variables in the query are computed, too. This means that in a logic program, input and output variables are not fixed.

3. Logic programs are especially well suited for applications in artificial intelligence. For example, logic programming is particularly well suited for the implementation of expert systems, where the rules of the program represent the knowledge of the experts. Further main application areas are deductive data bases and rapid prototyping.

The structure of the lecture is as follows: Since logic programs consist of formulas of predicate logic, and since proofs in predicate logic are used to execute logic programs, in Chapter 2 the required basics of predicate logic are presented. In Chapter 3 we introduce the proof principle of *resolution*, which is used in logic programming. Chapter 4 then considers the syntax, semantics, and expressiveness of (pure) logic programs as above and discusses the strategy for the execution of logic programs. In Chapter 5 we will then present the programming language Prolog and in particular consider those properties of Prolog which go beyond pure logic programs. Finally, in Chapter 6 we introduce an extension of logic programs by constraints.

I thank Peter Schneider-Kamp and René Thiemann for their constructive comments and suggestions while proofreading these notes and Denise Fromme for her work on the translation of the notes from German to English.

Chapter 2

Basics of Predicate Logic

In this chapter we will introduce the language of first-order predicate logic, which is used to formulate logic programs. To this end, we will recapitulate the syntax and semantics of predicate logic in Sections 2.1 and 2.2. In particular, this is also needed to introduce the notation used in the following. To execute a logic program, one has to analyze whether a formula (the *query*) is entailed by a set of formulas (the *facts* and *rules* of the *program*).

2.1 Syntax of Predicate Logic

The *syntax* states which words belong to a language. We begin with defining the alphabet used for the language of predicate logic.

Definition 2.1.1 (Signature) *A* signature (Σ, Δ) *is a pair with* $\Sigma = \bigcup_{n\in\mathbb{N}} \Sigma_n$ *and* $\Delta = \bigcup_{n\in\mathbb{N}} \Delta_n$. *Here,* Σ *and* Δ *are the unions of pairwise disjoint sets* Σ_n *and* Δ_n. *Every* $f \in \Sigma_n$ *is a* function symbol *of arity* n *and every* $p \in \Delta_n$ *is a* predicate symbol *of arity* n. *The elements of* Σ_0 *are also called* constants. *We always require* $\Sigma_0 \neq \varnothing$.

Example 2.1.2 *As an example, consider the following signature* (Σ, Δ) *with* $\Sigma = \Sigma_0 \cup \Sigma_3$ *and* $\Delta = \Delta_1 \cup \Delta_2$. *It corresponds to the signature of the logic program from Chapter 1, where* Σ *additionally contains the function symbol* date *of arity* 3 *and* Δ *contains the predicate symbol* born *of arity* 2.

$$
\begin{aligned}
\Sigma_0 &= \mathbb{N} \cup \{\mathsf{monika}, \mathsf{karin}, \mathsf{renate}, \mathsf{susanne}, \mathsf{aline}, \mathsf{werner}, \mathsf{klaus}, \mathsf{gerd}, \mathsf{peter}, \mathsf{dominique}\} \\
\Sigma_3 &= \{\mathsf{date}\} \\
\Delta_1 &= \{\mathsf{female}, \mathsf{male}, \mathsf{human}\} \\
\Delta_2 &= \{\mathsf{married}, \mathsf{motherOf}, \mathsf{fatherOf}, \mathsf{parent}, \mathsf{ancestor}, \mathsf{born}\}
\end{aligned}
$$

Now we define how data objects are represented in the language of predicate logic.

Definition 2.1.3 (Terms) *Let* (Σ, Δ) *be a signature and* $\mathcal{V}$ *be a set of* variables *such that* $\mathcal{V} \cap \Sigma = \varnothing$. *Then* $\mathcal{T}(\Sigma, \mathcal{V})$ *denotes the set of* terms *(over* Σ *and* $\mathcal{V}$*). Here,* $\mathcal{T}(\Sigma, \mathcal{V})$ *is the smallest set with*

- $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ *and*
- $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$, *if* $f \in \Sigma_n$ *and* $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$ *for some* $n \in \mathbb{N}$.

$\mathcal{T}(\Sigma)$ *stands for* $\mathcal{T}(\Sigma, \varnothing)$, *i.e., for the set of variable-free terms (or* ground terms*). For a term* t, $\mathcal{V}(t)$ *is the set of all variables in* t.

To distinguish variables from function and predicate symbols (in particular from constants), as in Prolog we use the convention that variables begin with upper case letters and function and predicate symbols start with lower case letters.

Example 2.1.4 *We again consider the signature* Σ *from Example 2.1.2. If* $\mathcal{V} = \{X, Y, Z, Mom, Grandma, \ldots\}$, *then we have the following terms in* $\mathcal{T}(\Sigma, \mathcal{V})$*:* monika, 42, date(15, 10, 1966), X, date(X, $Grandma$, date(Y, monika, 101)), ...

Now we can define how statements can be formed in the language of predicate logic.

Definition 2.1.5 (Formulas) *Let* (Σ, Δ) *be a signature and* $\mathcal{V}$ *be a set of variables. The set of* atomic formulas *over* (Σ, Δ) *and* $\mathcal{V}$ *is defined as* $\mathcal{A}t(\Sigma, \Delta, \mathcal{V}) = \{p(t_1, \ldots, t_n) \mid p \in \Delta_n \text{ for some } n,\ t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})\}$.

$\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ *is the set of all* formulas *over* (Σ, Δ) *and* $\mathcal{V}$. *Here,* $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ *is the smallest set with*

- $\mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
- *if* $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, *then* $\neg\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
- *if* $\varphi_1, \varphi_2 \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, *then* $(\varphi_1 \wedge \varphi_2), (\varphi_1 \vee \varphi_2), (\varphi_1 \to \varphi_2), (\varphi_1 \leftrightarrow \varphi_2) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
- *if* $X \in \mathcal{V}$ *and* $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, *then* $(\forall X\ \ \varphi), (\exists X\ \ \varphi) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

For a formula φ, $\mathcal{V}(\varphi)$ *is the set of all variables in* φ. *A variable* X *is* free *in a formula* φ *iff*

- φ *is an atomic formula and* $X \in \mathcal{V}(\varphi)$ *or*
- $\varphi = \neg\varphi_1$ *and* X *is free in* φ_1 *or*
- $\varphi = (\varphi_1 \cdot \varphi_2)$ *with* $\cdot \in \{\wedge, \vee, \to, \leftrightarrow\}$ *and* X *is free in* φ_1 *or in* φ_2 *or*
- $\varphi = (Q\, Y\ \varphi_1)$ *with* $Q \in \{\forall, \exists\}$, X *is free in* φ_1 *and* $X \neq Y$.

A formula φ *is* closed *iff no free variables occur in* φ. *A formula is* quantifier-free *iff it does not contain the characters* $\forall$ *or* $\exists$.

So $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ is a formal language in which mathematical facts but also facts of daily life can be formulated. In the following we will usually omit unnecessary brackets (i.e., we write "$\forall X$ human(X)" instead of "($\forall X$ human(X))").

Example 2.1.6 *Let (Σ, Δ) again be the signature from Example 2.1.2. Then we can construct the following formulas.*

$$\mathsf{female}(\mathsf{monika}) \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \quad (2.1)$$

$$\mathsf{motherOf}(X, \mathsf{susanne}) \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \quad (2.2)$$

$$\mathsf{born}(\mathsf{monika}, \mathsf{date}(15, 10, 1966)) \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \quad (2.3)$$

$$\forall F\ (\mathsf{married}(\mathsf{gerd}, F) \wedge \mathsf{motherOf}(F, K)) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V}) \quad (2.4)$$

$$\mathsf{married}(\mathsf{gerd}, F) \wedge \neg(\forall F\ \mathsf{motherOf}(F, K)) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V}). \quad (2.5)$$

In the formula in Line (2.2), the variable X is free. For the formula φ in Line (2.4), we have $\mathcal{V}(\varphi) = \{F, K\}$ but only the variable K is free here. In the formula in Line (2.5), F as well as K are free.

We abbreviate formulas of the form "$\forall X_1\ (\ldots(\forall X_n\ \varphi)\ldots)$" by "$\forall X_1, \ldots, X_n\ \varphi$". Analogously, "$\exists X_1, \ldots, X_n\ \varphi$" is an abbreviation for "$\exists X_1\ (\ldots(\exists X_n\ \varphi)\ldots)$".

Example 2.1.7 *The logic program from Chapter 1 corresponds to the following set of formulas over the signature from Example 2.1.2. As stated above, all variables in logic programs are universally quantified:*

$$\begin{array}{lll}
 & & \mathsf{female}(\mathsf{monika}) \\
 & & \mathsf{female}(\mathsf{karin}) \\
 & & \mathsf{female}(\mathsf{renate}) \\
 & & \mathsf{female}(\mathsf{susanne}) \\
 & & \mathsf{female}(\mathsf{aline}) \\
 & & \\
 & & \mathsf{male}(\mathsf{werner}) \\
 & & \mathsf{male}(\mathsf{klaus}) \\
 & & \mathsf{male}(\mathsf{gerd}) \\
 & & \mathsf{male}(\mathsf{peter}) \\
 & & \mathsf{male}(\mathsf{dominique}) \\
 & & \\
 & & \mathsf{married}(\mathsf{werner}, \mathsf{monika}) \\
 & & \mathsf{married}(\mathsf{gerd}, \mathsf{renate}) \\
 & & \mathsf{married}(\mathsf{klaus}, \mathsf{susanne}) \\
 & & \\
 & & \mathsf{motherOf}(\mathsf{monika}, \mathsf{karin}) \\
 & & \mathsf{motherOf}(\mathsf{monika}, \mathsf{klaus}) \\
 & & \mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}) \\
 & & \mathsf{motherOf}(\mathsf{renate}, \mathsf{peter}) \\
 & & \mathsf{motherOf}(\mathsf{susanne}, \mathsf{aline}) \\
 & & \mathsf{motherOf}(\mathsf{susanne}, \mathsf{dominique}) \\
 & & \\
\forall X & & \mathsf{human}(X) \\
 & & \\
\forall V, F, K & \mathsf{married}(V, F) \wedge \mathsf{motherOf}(F, K) \rightarrow & \mathsf{fatherOf}(V, K)
\end{array}$$

$$\begin{array}{lrcl}
\forall X, Y & \mathsf{motherOf}(X, Y) & \rightarrow & \mathsf{parent}(X, Y) \\
\forall X, Y & \mathsf{fatherOf}(X, Y) & \rightarrow & \mathsf{parent}(X, Y) \\
 & & & \\
\forall V, X & \mathsf{parent}(V, X) & \rightarrow & \mathsf{ancestor}(V, X) \\
\forall V, Y, X & \mathsf{parent}(V, Y) \wedge \mathsf{ancestor}(Y, X) & \rightarrow & \mathsf{ancestor}(V, X)
\end{array}$$

Finally, we introduce the concept of substitution. Substitutions allow the replacement (or "instantiation") of variables by terms.

Definition 2.1.8 (Substitution) *A mapping $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ is a* substitution *iff $\sigma(X) \neq X$ holds for only finitely many $X \in \mathcal{V}$. $DOM(\sigma) = \{X \in \mathcal{V} \mid \sigma(X) \neq X\}$ is the* domain *of σ and $RANGE(\sigma) = \{\sigma(X) \mid X \in DOM(\sigma)\}$ is the* range *of σ. Since $DOM(\sigma)$ is finite, a substitution σ can be represented as the finite set of pairs $\{X/\sigma(X) \mid X \in DOM(\sigma)\}$. A substitution σ is a* ground substitution *on $DOM(\sigma)$ iff $\mathcal{V}(\sigma(X)) = \varnothing$ for all $X \in DOM(\sigma)$.*[1] *A substitution σ is a* variable renaming *iff σ is injective and $\sigma(X) \in \mathcal{V}$ for all $X \in \mathcal{V}$.*

Substitutions are extended to mappings $\sigma : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{T}(\Sigma, \mathcal{V})$ on terms in a homomorphic way, i.e., $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. In an analogous way, substitutions can also be applied to formulas:

(1) $\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))$

(2) $\sigma(\neg\varphi) = \neg\sigma(\varphi)$

(3) $\sigma(\varphi_1 \cdot \varphi_2) = \sigma(\varphi_1) \cdot \sigma(\varphi_2)$ *for* $\cdot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$

(4) $\sigma(Q\, X\ \varphi) = Q\, X\ \sigma(\varphi)$, *for* $Q \in \{\forall, \exists\}$, *if* $X \notin \mathcal{V}(RANGE(\sigma)) \cup DOM(\sigma)$

(5) $\sigma(Q\, X\ \varphi) = Q\, X'\ \sigma(\delta(\varphi))$, *for* $Q \in \{\forall, \exists\}$, *if* $X \in \mathcal{V}(RANGE(\sigma)) \cup DOM(\sigma)$. *Here, X' is a fresh variable with $X' \notin \mathcal{V}(RANGE(\sigma)) \cup DOM(\sigma) \cup \mathcal{V}(\varphi)$ and $\delta = \{X/X'\}$.*

The condition in Case (4) means that the application of σ to the formula $\forall X\ \varphi$ is only possible if the variable X bound by the quantifier is not replaced and if no other variable in φ is replaced by a term that contains X. Otherwise, X must be renamed to a variable X' (in Case (5)) and afterwards the substitution σ is applied. Such a renaming of quantified variables is called bound renaming.

An instance $\sigma(t)$ of a term t (an instance $\sigma(\varphi)$ of a quantifier-free formula φ, respectively) is a ground instance *iff $\mathcal{V}(\sigma(t)) = \varnothing$ ($\mathcal{V}(\sigma(\varphi)) = \varnothing$, respectively).*

Example 2.1.9 *We again consider the signature of Example 2.1.2. Here, an example for a substitution is $\sigma = \{X/\mathsf{date}(X, Y, Z), Y/\mathsf{monika}, Z/\mathsf{date}(Z, Z, Z)\}$. We get*

$$\begin{array}{rcl}
\sigma(\mathsf{date}(X, Y, Z)) & = & \mathsf{date}(\mathsf{date}(X, Y, Z), \mathsf{monika}, \mathsf{date}(Z, Z, Z)) \\
\sigma(\forall Y\ \mathsf{married}(X, Y)) & = & \forall Y'\ \mathsf{married}(\mathsf{date}(X, Y, Z), Y')
\end{array}$$

Instead of "$\sigma(\mathsf{date}(X, Y, Z))$" we often write

$$\mathsf{date}(X, Y, Z)\ [X/\mathsf{date}(X, Y, Z),\ Y/\mathsf{monika},\ Z/\mathsf{date}(Z, Z, Z)].$$

[1] We often just speak of a "ground substitution" and assume that $DOM(\sigma)$ is chosen large enough to map all considered variables to ground terms.

2.2 Semantics of Predicate Logic

Our goal is to formally represent statements about the world with formulas from $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$. So far we only defined how formulas in $\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ are built. However, we have not defined what such formulas *mean*. For this, we have to assign a *semantics* to those formulas. Then we can also define which formulas (i.e., which *queries*) are entailed by a given set of formulas (i.e., by a *logic program*).

For defining the semantics, so-called *interpretations* are used. They fix a set of objects $\mathcal{A}$, they assign a function α_f to every (syntactic) function symbol f, they assign a relation (or subset) α_p to every (syntactic) predicate symbol p, and they map every variable $X \in \mathcal{V}$ to an object from $\mathcal{A}$. An interpretation maps every term to an object from $\mathcal{A}$ and every formula is true or false with respect to an interpretation.

Definition 2.2.1 (Interpretation, Structure, Satisfiability, Model) *For a signature (Σ, Δ), an* interpretation *is a triple $I = (\mathcal{A}, \alpha, \beta)$. The set $\mathcal{A}$ is the* carrier *of the interpretation, where we require $\mathcal{A} \neq \varnothing$. Moreover, α maps every function symbol $f \in \Sigma_n$ to a function $\alpha_f : \mathcal{A}^n \to \mathcal{A}$, and every predicate symbol $p \in \Delta_n$ with $n \geq 1$ to a set (relation) $\alpha_p \subseteq \mathcal{A}^n$. For $p \in \Delta_0$, $\alpha_p \in \{TRUE, FALSE\}$ holds. The function α_f resp. the relation α_p are the* meaning *of the function symbol f resp. of the predicate symbol p under the interpretation I. The mapping $\beta : \mathcal{V} \to \mathcal{A}$ is called a* variable assignment *for the interpretation I.*

For every interpretation I we get a function $I : \mathcal{T}(\Sigma, \mathcal{V}) \to \mathcal{A}$ as follows:

$$\begin{aligned} I(X) &= \beta(X) \text{ for all } X \in \mathcal{V} \\ I(f(t_1, \ldots, t_n)) &= \alpha_f(I(t_1), \ldots, I(t_n)) \text{ for all } f \in \Sigma_n \text{ and } t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V}) \end{aligned}$$

$I(t)$ is called the interpretation of the term *t under the interpretation I.*

For $X \in \mathcal{V}$ and $\mathbf{a} \in \mathcal{A}$, $\beta[\![X/\mathbf{a}]\!]$ is the variable assignment with $\beta[\![X/\mathbf{a}]\!](X) = \mathbf{a}$ and $\beta[\![X/\mathbf{a}]\!](Y) = \beta(Y)$ for all $Y \in \mathcal{V}$ with $Y \neq X$. $I[\![X/\mathbf{a}]\!]$ is the interpretation $(\mathcal{A}, \alpha, \beta[\![X/\mathbf{a}]\!])$.

An interpretation $I = (\mathcal{A}, \alpha, \beta)$ satisfies *a formula $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, written as "$I \models \varphi$", iff*

$$\begin{array}{lrll} & \varphi &= p(t_1, \ldots, t_n) & \text{and } p \in \Delta_n \text{ with } (I(t_1), \ldots, I(t_n)) \in \alpha_p \text{ for } n \geq 1 \\ \text{or} & \varphi &= p & \text{and } p \in \Delta_0 \text{ with } \alpha_p = TRUE \\ \text{or} & \varphi &= \neg\varphi_1 & \text{and } I \not\models \varphi_1 \\ \text{or} & \varphi &= \varphi_1 \wedge \varphi_2 & \text{and } I \models \varphi_1 \text{ and } I \models \varphi_2 \\ \text{or} & \varphi &= \varphi_1 \vee \varphi_2 & \text{and } I \models \varphi_1 \text{ or } I \models \varphi_2 \\ \text{or} & \varphi &= \varphi_1 \rightarrow \varphi_2 & \text{and if } I \models \varphi_1, \text{ then also } I \models \varphi_2 \\ \text{or} & \varphi &= \varphi_1 \leftrightarrow \varphi_2 & \text{and } I \models \varphi_1 \text{ iff } I \models \varphi_2 \\ \text{or} & \varphi &= \forall X\, \varphi_1 & \text{and } I[\![X/\mathbf{a}]\!] \models \varphi_1 \text{ for all } \mathbf{a} \in \mathcal{A} \\ \text{or} & \varphi &= \exists X\, \varphi_1 & \text{and } I[\![X/\mathbf{a}]\!] \models \varphi_1 \text{ for some } \mathbf{a} \in \mathcal{A} \end{array}$$

An interpretation I is a model *of φ iff $I \models \varphi$ holds. I is a model of a set of formulas Φ ("$I \models \Phi$") iff $I \models \varphi$ holds for all $\varphi \in \Phi$. Two formulas φ_1 and φ_2 are* equivalent *iff for all interpretations I we have $I \models \varphi_1$ iff $I \models \varphi_2$.*

A formula (resp. a set of formulas) is satisfiable *iff it has a model, and it is* unsatisfiable *iff it does not have a model. It is* valid *iff every interpretation is a model.*

An interpretation without a variable assignment $S = (\mathcal{A}, \alpha)$ is called a structure.[2] *Thus, a structure has a carrier set, where we assign functions from this carrier set to the function symbols in Σ and we assign relations from the carrier set to the predicate symbols in Δ.*

If we only consider closed formulas, we can define the notions of satisfiability *and* models *also for structures. A structure S satisfies a closed formula φ (i.e., "$S \models \varphi$" resp. S is a model of φ) iff $I \models \varphi$ for an interpretation of the form $I = (\mathcal{A}, \alpha, \beta)$. The variable assignment β is irrelevant here, since φ does not contain any free variables. Similarly, we can define $S(t)$ for ground terms t (the interpretation of term t under the structure S).*

Example 2.2.2 *We again regard the signature from Example 2.1.2. For instance, an interpretation for this signature is $I = (\mathcal{A}, \alpha, \beta)$ with*

$$\begin{aligned}
\mathcal{A} &= \mathbb{N} \\
\alpha_n &= n \text{ for all } n \in \mathbb{N} \\
\alpha_{\mathsf{monika}} &= 0 \\
\alpha_{\mathsf{karin}} &= 1 \\
\alpha_{\mathsf{renate}} &= 2 \\
&\vdots \\
\alpha_{\mathsf{date}}(n_1, n_2, n_3) &= n_1 + n_2 + n_3 \text{ for all } n_1, n_2, n_3 \in \mathbb{N} \\
\alpha_{\mathsf{female}} &= \{n \mid n \text{ is even}\} \\
\alpha_{\mathsf{male}} &= \{n \mid n \text{ is odd}\} \\
\alpha_{\mathsf{human}} &= \mathbb{N} \\
\alpha_{\mathsf{married}} &= \{(n, m) \mid n > m\} \\
&\vdots \\
\beta(X) &= 0 \\
\beta(Y) &= 1 \\
\beta(Z) &= 2 \\
&\vdots
\end{aligned}$$

Then $I(\mathsf{date}(\mathsf{1}, X, \mathsf{karin})) = \alpha_{\mathsf{date}}(\alpha_{\mathsf{1}}, \beta(X), \alpha_{\mathsf{karin}}) = 1 + 0 + 1 = 2$. Hence, we have

$$I \models \mathsf{married}(\mathsf{date}(\mathsf{1}, X, \mathsf{karin}), \mathsf{karin}).$$

Moreover, we have

$$I \models \forall X\ \mathsf{female}(\mathsf{date}(X, X, \mathsf{monika})),$$

since for all $\mathbf{a} \in \mathcal{A}$, $I[\![X/\mathbf{a}]\!](\mathsf{date}(X, X, \mathsf{monika})) = \mathbf{a} + \mathbf{a} + 0$ is an even number. Since the formula $\forall X\ \mathsf{female}(\mathsf{date}(X, X, \mathsf{monika}))$ is closed, the structure $S = (\mathcal{A}, \alpha)$ is already a model of the formula, i.e.,

$$S \models \forall X\ \mathsf{female}(\mathsf{date}(X, X, \mathsf{monika})).$$

[2] If there are no predicate symbols except for equality, it is also called an *algebra*.

All of the above formulas are therefore satisfiable, but they are not valid, since they are not satisfied by every interpretation. Examples for valid formulas are $\varphi \vee \neg\varphi$ for all formulas φ. Examples for unsatisfiable formulas are $\varphi \wedge \neg\varphi$ for all formulas φ.

The connection between the syntactic concept "substitution" and the semantic concept "variable assignment" is described by the following lemma.

Lemma 2.2.3 (Substitution Lemma) *Let $I = (\mathcal{A}, \alpha, \beta)$ be an interpretation for a signature (Σ, Δ), let $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$ be a substitution. Then we have:*

(a) $I(\sigma(t)) = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](t)$ for all $t \in \mathcal{T}(\Sigma, \mathcal{V})$

(b) $I \models \sigma(\varphi)$ iff $I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!] \models \varphi$ for all $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

Proof.

(a) The proof is by structural induction on t, where the induction corresponds to the definition of the data structure for terms. In the induction base t is either a variable or a constant (i.e., a function symbol in Σ_0).

If t is a variable X_i, then

$$I(\sigma(X_i)) = I(t_i) = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](X_i).$$

If t is a variable Y that is different from all $X_1, \ldots, X_n$, we get

$$I(\sigma(Y)) = I(Y) = \beta(Y) = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](Y).$$

If t is of the form $f(s_1, \ldots, s_k)$ (where $k = 0$ is possible), we have

$$I(\sigma(t)) = I(\sigma(f(s_1, \ldots, s_k))) = \alpha_f(I(\sigma(s_1)), \ldots, I(\sigma(s_k))).$$

The induction hypothesis implies $I(\sigma(s_i)) = I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](s_i)$. Therefore we get

$$\begin{aligned}
& \alpha_f(I(\sigma(s_1)), \ldots, I(\sigma(s_k))) \\
= \; & \alpha_f(I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](s_1), \ldots, I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](s_k)) \\
= \; & I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](f(s_1, \ldots, s_k)) \\
= \; & I[\![X_1/I(t_1), \ldots, X_n/I(t_n)]\!](t).
\end{aligned}$$

(b) We use structural induction on the structure of the formula φ.

If $\varphi = p(s_1, \ldots, s_m)$, we have

$$\begin{aligned}
I \models \sigma(\varphi) \quad & \text{iff} \quad I \models p(\sigma(s_1), \ldots, \sigma(s_m)) \\
& \text{iff} \quad (I(\sigma(s_1)), \ldots, I(\sigma(s_m))) \in \alpha_p \\
& \text{iff} \quad (I[\![X_1/I(s_1), \ldots, X_n/I(s_m)]\!](s_1), \ldots, \\
& \qquad I[\![X_1/I(s_1), \ldots, X_n/I(s_m)]\!](s_m)) \in \alpha_p
\end{aligned}$$

by (a). Therefore we get

$$\begin{array}{ll} & (I[\![X_1/I(s_1),\ldots,X_n/I(s_m)]\!](s_1),\ldots,I[\![X_1/I(s_1),\ldots,X_n/I(s_m)]\!](s_m)) \in \alpha_p \\ \text{iff} & I[\![X_1/I(s_1),\ldots,X_n/I(s_m)]\!] \models p(s_1,\ldots,s_m) \\ \text{iff} & I[\![X_1/I(s_1),\ldots,X_n/I(s_m)]\!] \models \varphi. \end{array}$$

The cases $\varphi = \neg\varphi_1$ or $\varphi = \varphi_1 \cdot \varphi_2$ with $\cdot \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ are easy. Let us look at the case $\varphi = \forall X\ \varphi_1$ (the case $\varphi = \exists X\ \varphi_1$ works analogously). W.l.o.g. we only consider the case $X \notin \mathcal{V}(RANGE(\sigma)) \cup DOM(\sigma)$. (Otherwise we convert the formula first into $\forall X'\ \varphi_1[X/X']$, which is obviously equivalent to φ.) We have $\sigma(\varphi) = \forall X\ \sigma(\varphi_1)$ and

$$I \models \sigma(\varphi) \quad \text{iff} \quad I[\![X/\mathbf{a}]\!] \models \sigma(\varphi_1) \text{ for all } \mathbf{a} \in \mathcal{A}.$$

Let $\sigma = \{X_1/t_1,\ldots,X_n/t_n\}$ (where by assumption X is not one of the variables X_i). By the induction hypothesis we obtain

$$\begin{array}{ll} & I[\![X/\mathbf{a}]\!] \models \sigma(\varphi_1) \text{ for all } \mathbf{a} \in \mathcal{A} \\ \text{iff} & I[\![X/\mathbf{a}]\!][\![X_1/I(t_1),\ldots,X_n/I(t_n)]\!] \models \varphi_1 \text{ for all } \mathbf{a} \in \mathcal{A}. \end{array}$$

Since X is pairwise different from all $X_1,\ldots,X_n$, we have

$$I[\![X/\mathbf{a}]\!][\![X_1/I(t_1),\ldots,X_n/I(t_n)]\!] = I[\![X_1/I(t_1),\ldots,X_n/I(t_n)]\!][\![X/\mathbf{a}]\!]$$

and therefore, we get

$$\begin{array}{ll} & I[\![X_1/I(t_1),\ldots,X_n/I(t_n)]\!][\![X/\mathbf{a}]\!] \models \varphi_1 \text{ for all } \mathbf{a} \in \mathcal{A} \\ \text{iff} & I[\![X_1/I(t_1),\ldots,X_n/I(t_n)]\!] \models \forall X\ \varphi_1 \\ \text{iff} & I[\![X_1/I(t_1),\ldots,X_n/I(t_n)]\!] \models \varphi. \end{array}$$

□

Example 2.2.4 *We consider the interpretation I from Example 2.2.2, the substitution $\sigma = \{X/\mathsf{date}(1, X, \mathsf{karin})\}$, and the term $t = \mathsf{date}(X, Y, Z)$. Then we have*

$$\begin{array}{rcl} I(\sigma(t)) & = & I(\mathsf{date}(\mathsf{date}(1, X, \mathsf{karin}), Y, Z)) \\ & = & \alpha_1 + \beta(X) + \alpha_{\mathsf{karin}} + \beta(Y) + \beta(Z) \\ & = & 1 + 0 + 1 + 1 + 2 \\ & = & 5. \end{array}$$

Similarly, we have

$$\begin{array}{rcl} I[\![X/I(\mathsf{date}(1, X, \mathsf{karin}))]\!](t) & = & I[\![X/2]\!](\mathsf{date}(X, Y, Z)) \\ & = & 2 + \beta(Y) + \beta(Z) \\ & = & 2 + 1 + 2 \\ & = & 5. \end{array}$$

Now we define when a formula is entailed by a set of formulas. This is the question that is analyzed during the execution of logic programs.

Definition 2.2.5 (Entailment) *A set of formulas* Φ entails *the formula* φ *(abbreviated "*$\Phi \models \varphi$*") iff for all interpretations* I *with* $I \models \Phi$, $I \models \varphi$ *holds. If* Φ *and* φ *have no free variables, then this is equivalent to the requirement that for all structures* S *with* $S \models \Phi$, $S \models \varphi$ *holds. (The symbol "*$\models$*" thus stands for satisfiability by an interpretation as well as entailment by a set of formulas. Which of those two is meant depends on whether an interpretation or a set of formulas is on the left of the symbol "*$\models$*".) Instead of "*$\varnothing \models \varphi$*" we usually write "*$\models \varphi$*" (i.e., the formula* φ *is valid).*

Example 2.2.6 *Let* Φ *be the set of formulas from Example 2.1.7 that corresponds to the logic program in Chapter 1. The query*

```
?- male(gerd).
```

means that one has to prove $\Phi \models \mathsf{male}(\mathsf{gerd})$. *This clearly holds in the example above, since the formula* $\mathsf{male}(\mathsf{gerd})$ *is contained in* Φ. *We also have* $\Phi \models \mathsf{human}(\mathsf{gerd})$, *since* Φ *contains the formula* $\forall X\ \mathsf{human}(X)$.

The query

```
?- motherOf(X,susanne).
```

means that one has to prove $\Phi \models \exists X\ \mathsf{motherOf}(X, \mathsf{susanne})$. *As mentioned, variables* X *are universally quantified in the logic program and existentially quantified in queries. Since* Φ *contains the formula* $\mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne})$, $\Phi \models \exists X\ \mathsf{motherOf}(X, \mathsf{susanne})$ *holds. Here, the variable* X *needs to be assigned the meaning of* renate.

Thus, for executing logic programs one has to figure out if $\Phi \models \varphi$ holds for a set of formulas Φ (the logic program) and a formula φ (the query). In the following chapter we will show how this question can be analyzed automatically.

Chapter 3

Resolution

The notion of "entailment" is defined semantically. This definition is not suitable for automation, because to analyze $\Phi \models \varphi$, we would need to check all (infinitely many) interpretations. For every interpretation we would need to find out whether it is a model of Φ and in this case check if it is also a model of φ.

To analyze entailment in a syntactic way (that is amenable to automation) instead, we use a so-called *calculus*. A calculus consists of certain (usually purely syntactic) rules, which allow us to derive new formulas from a set of formulas Φ. Thus, checking whether φ can be *derived* from Φ can be done in a syntactic manner.

A calculus must be *sound* in order to use to infer statements about entailment, i.e., if φ can be derived from Φ, then the entailment $\Phi \models \varphi$ should hold as well. If the other direction also holds, i.e., if the entailment $\Phi \models \varphi$ implies that φ can be derived from Φ, then the calculus is *complete*.

In this chapter, we will present the so-called *resolution calculus*, which is used in logic programming to analyze $\Phi \models \varphi$. We show that this calculus both sound and complete, i.e., here, entailment and derivability correspond to each other.

The idea behind the resolution calculus is that the entailment problem $\Phi \models \varphi$ can be reduced to an *unsatisfiability problem* . Then, this unsatisfiability problem can be analyzed by resolution. The following lemma shows how every entailment problem can be transformed into an unsatisfiability problem.

Lemma 3.0.1 (From Entailment to Unsatisfiability)
Let $\varphi_1, \ldots, \varphi_k, \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$. Then we have $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ iff the formula $\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi$ is unsatisfiable.

Proof.

We have	$\{\varphi_1, \ldots, \varphi_k\} \models \varphi$
iff	for all interpretations I with $I \models \{\varphi_1, \ldots, \varphi_k\}$ we have $I \models \varphi$
iff	there is no interpretation I with $I \models \{\varphi_1, \ldots, \varphi_k\}$ and $I \models \neg\varphi$
iff	$\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi$ is unsatisfiable

□

Example 3.0.2 *To show that in the logic program with the fact*

```
motherOf(renate,susanne).
```

the query

```
?- motherOf(X,susanne).
```

can be proved, one has to show that

$$\{\mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne})\} \models \exists\, X\ \mathsf{motherOf}(X, \mathsf{susanne})$$

holds. Instead, one can show unsatisfiability of the following formula:

$$\mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \exists\, X\ \mathsf{motherOf}(X, \mathsf{susanne})$$

In general, the problem of showing unsatisfiability of a set of formulas (and therefore also the problem of analyzing entailment) is *undecidable*, see, e.g., [Gra11, Satz 4.30]. Thus, there is no algorithm that always terminates and can check unsatisfiability for every formula. But unsatisfiability (and therefore also entailment) is *semi-decidable*. Thus, there is an algorithm that determines unsatisfiability in finite time for every unsatisfiable formula, but it might not terminate for satisfiable formulas. For the entailment problem this means that this algorithm can prove $\Phi \models \varphi$ in finite time whenever it holds. But if $\Phi \not\models \varphi$ holds, then the algorithm might not terminate. Since the resolution calculus is a sound and complete method for checking unsatisfiability of a formula, it is therefore such a semi-decision algorithm. If the formula is unsatisfiable, then this can also be shown by an automation of the resolution calculus. But if it is satisfiable, then the automation of the resolution calculus might not terminate.

We introduce the resolution principle for showing unsatisfiability of a formula φ in four steps. The basic idea is to reduce the resolution principle for predicate logic to the resolution principle for propositional logic. In propositional logic, the soundness and completeness of resolution is easier to prove. (Moreover, in propositional logic, resolution is a *decision algorithm* for unsatisfiability.)

1. First we show that φ can be transformed into *Skolem normal form* to verify unsatisfiability (Section 3.1). Formulas in Skolem normal form are closed, they do not contain any existential quantifiers, and they only contain universal quantifiers on the outside. Thus, such formulas are of the form $\forall X_1, \ldots, X_n\ \psi$, where ψ is quantifier-free and does not contain any variables except $X_1, \ldots, X_n$.

2. Then we show that we do not need to consider *all* interpretations to check unsatisfiability for formulas in Skolem normal form, but it suffices to restrict ourselves to so-called *Herbrand interpretations* (Section 3.2). These are interpretations where ground terms are interpreted as "themselves". This results in a first algorithm to analyze unsatisfiability, which however is still quite inefficient.

3. To get a more efficient algorithm, we extend propositional resolution (Section 3.3) to predicate logic by using *unification* (Section 3.4).

4. To increase efficiency even further, we show in Section 3.5 how resolution can be restricted further.

3.1 Skolem Normal Form

As mentioned before, the goal is to simplify formulas into a normal form of the form $\forall X_1, \ldots, X_n\ \psi$, where ψ is quantifier-free and does not contain any variables except $X_1, \ldots, X_n$. Here, we proceed in two steps. First the formula is transformed into *prenex normal form.*

Definition 3.1.1 (Prenex Normal Form) *A formula φ is in* prenex normal form *iff it has the form $Q_1\, X_1\ \ldots Q_n\, X_n\ \psi$ with $Q_i \in \{\forall, \exists\}$ and where ψ is quantifier-free.*

The following theorem shows that (and how) every formula can be transformed into an equivalent formula in prenex normal form.

Theorem 3.1.2 (Transformation Into Prenex Normal Form) *For every formula φ one can automatically construct an equivalent formula φ' in prenex normal form.*

Proof. First, all subformulas $\varphi_1 \leftrightarrow \varphi_2$ in φ are replaced by $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$. Then, all subformulas $\varphi_1 \rightarrow \varphi_2$ in φ are replaced by $\neg\varphi_1 \vee \varphi_2$. The transformation of the remaining formula is done by the following algorithm $PRENEX$, whose termination and soundness is obvious. The input of the algorithm is an arbitrary formula φ without the Boolean operators "$\leftrightarrow$" and "$\rightarrow$", and its output is an equivalent formula in prenex normal form.

- If φ is quantifier-free, then return φ.
- If $\varphi = \neg\varphi_1$, then compute $PRENEX(\varphi_1) = Q_1\, X_1\ \ldots Q_n\, X_n\ \psi$. Afterwards, return $\overline{Q_1}\, X_1\ \ldots \overline{Q_n}\, X_n\ \neg\psi$, where $\overline{\forall} = \exists$ and $\overline{\exists} = \forall$.
- If $\varphi = \varphi_1 \cdot \varphi_2$ with $\cdot \in \{\wedge, \vee\}$, then compute $PRENEX(\varphi_1) = Q_1\, X_1\ \ldots Q_n\, X_n\ \psi_1$ and $PRENEX(\varphi_2) = R_1\, Y_1\ \ldots R_m\, Y_m\ \psi_2$. By renaming bound variables, we ensure that $X_1, \ldots, X_n$ do not occur in $R_1\, Y_1\ \ldots R_m\, Y_m\ \psi_2$ and that $Y_1, \ldots, Y_m$ do not occur in $Q_1\, X_1\ \ldots Q_n\, X_n\ \psi_1$. Now return the following formula:

$$Q_1\, X_1\ \ldots Q_n\, X_n\ R_1\, Y_1\ \ldots R_m\, Y_m\quad (\psi_1 \cdot \psi_2)$$

- If $\varphi = Q\, X\ \varphi_1$ with $Q \in \{\forall, \exists\}$, then compute the formula $PRENEX(\varphi_1) = Q_1\, X_1\ \ldots\ Q_n\, X_n\ \psi$. By renaming bound variables, we ensure that $X_1, \ldots, X_n$ are different from X. Then return $Q\, X\ Q_1\, X_1\ \ldots Q_n\, X_n\ \psi$.

□

Example 3.1.3 *We consider the transformation of the following formula:*

$$\neg\exists X\,(\,\mathsf{married}(X, Y) \vee \neg\exists Y\ \mathsf{motherOf}(X, Y)\,)$$

First we compute $PRENEX(\neg\exists Y\ \mathsf{motherOf}(X, Y)) = \forall Y\ \neg\mathsf{motherOf}(X, Y)$. To make the bound variable Y in this subformula different from the free variable Y in the subformula $\mathsf{married}(X, Y)$, we rename the former to Z and therefore get $\forall Z\ \neg\mathsf{motherOf}(X, Z)$.

Now we compute $PRENEX(\mathsf{married}(X,Y) \vee \neg\exists Y\ \mathsf{motherOf}(X,Y))$*, which yields the following formula:*

$$\forall Z\,(\mathsf{married}(X,Y) \vee \neg\mathsf{motherOf}(X,Z))$$

Finally, we obtain $PRENEX(\neg\exists X\,(\mathsf{married}(X,Y) \vee \neg\exists Y\ \mathsf{motherOf}(X,Y))) = \forall X\,\exists Z\,\neg(\mathsf{married}(X,Y) \vee \neg\mathsf{motherOf}(X,Z))$.

Example 3.1.4 *As another example, we consider the transformation of the formula* $\mathsf{motherOf}(\mathsf{renate},\mathsf{susanne}) \wedge \neg\exists\, X\ \mathsf{motherOf}(X,\mathsf{susanne})$ *from Example 3.0.2, whose unsatisfiability must be shown in order to prove the query*

```
?- motherOf(X,susanne).
```

in the logic program with the fact

```
motherOf(renate,susanne).
```

Here, by the transformation to prenex normal form we get the following formula:

$$\forall\, X\ \mathsf{motherOf}(\mathsf{renate},\mathsf{susanne}) \wedge \neg\mathsf{motherOf}(X,\mathsf{susanne})$$

Now we define the Skolem normal form.

Definition 3.1.5 (Skolem Normal Form) *A formula* φ *is in* Skolem normal form *iff it is closed and it has the form* $\forall\, X_1,\ldots,X_n\ \psi$*, where* ψ *is quantifier-free.*

To transform a formula into Skolem normal form, it is first transformed into prenex normal form. The remaining transformation is used to get rid of the free variables and the existential quantifiers. In contrast to prenex normal form, there is not an equivalent formula in Skolem normal form for every formula, because clearly there is no formula in Skolem normal form that is equivalent to $\mathsf{female}(X)$ or to $\exists X\ \mathsf{female}(X)$, for example. But for every formula there is a *satisfiability-equivalent* formula in Skolem normal form.

The idea of the transformation is to first create the prenex normal form, then bind all free variables by existential quantifiers, and finally eliminate all existentially quantified variables by using fresh function symbols.

Theorem 3.1.6 (Transformation into Skolem Normal Form) *For every formula* φ *one can automatically construct a formula* φ' *in Skolem normal form such that* φ *is satisfiable iff* φ' *is satisfiable.*

Proof. First, the formula φ is transformed into prenex normal form as in Theorem 3.1.2. Let $X_1,\ldots,X_n$ be the free variables in the resulting formula φ_1 that is equivalent to φ. Then φ_1 is transformed into the closed formula φ_2 of the form $\exists X_1,\ldots,X_n\ \varphi_1$ that is satisfiability-equivalent to φ_1: From $I \models \varphi_1$ we obtain $I[\![X_1/\beta(X_1),\ldots,X_n/\beta(X_n)]\!] \models \varphi_1$ and therefore obviously also $I \models \exists X_1,\ldots,X_n\ \varphi_1$. On the other hand, $I \models \exists X_1,\ldots,X_n\ \varphi_1$ implies that there exist $\mathbf{a_1},\ldots,\mathbf{a_n}$, such that $I[\![X_1/\mathbf{a_1},\ldots,X_n/\mathbf{a_n}]\!] \models \varphi_1$ holds.

The formula φ_2 is therefore closed, in prenex normal form, and satisfiability-equivalent to φ. Now we eliminate the existential quantifiers from the outside to the inside. If φ_2 is the formula $\forall X_1, \ldots, X_n \exists Y\ \psi$, then we replace φ_2 by the following formula:

$$\forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)]$$

Thus, here we substitute all occurrences of Y by $f(X_1, \ldots, X_n)$, where f is a fresh function symbol of arity n. This procedure is repeated until no existential quantifiers are left anymore. The resulting formula is satisfiability-equivalent to φ_2 and therefore also to φ. The reason is that:

$$\begin{array}{ll} & I \models \forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)] \\ \curvearrowright & I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!] \models \psi[Y/f(X_1, \ldots, X_n)] \text{ for all } \mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A} \\ \curvearrowright & I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!][\![Y/I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!](f(X_1, \ldots, X_n))]\!] \models \psi \\ & \qquad\qquad \text{for all } \mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}, \text{ by the Substitution Lemma 2.2.3} \\ \curvearrowright & I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!] \models \exists Y\ \psi \text{ for all } \mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A} \\ \curvearrowright & I \models \forall X_1, \ldots, X_n\ \exists Y\ \psi \text{ for all } \mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A} \end{array}$$

Note that $I \models \forall X_1, \ldots, X_n \exists Y\ \psi$ with $I = (\mathcal{A}, \alpha, \beta)$ means that for all $\mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}$ there is a $\mathbf{b} \in \mathcal{A}$ such that $I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}, Y/\mathbf{b}]\!] \models \psi$. Let F be the function from $\mathcal{A}^n \to \mathcal{A}$ that assigns to each tuple $(\mathbf{a_1}, \ldots, \mathbf{a_n})$ the corresponding $\mathbf{b}$. Let $I' = (\mathcal{A}, \alpha', \beta)$, where α' differs from α only in the interpretation of the new function symbol f. Here, we define $\alpha'_f = F$. Then we obtain $I' \models \forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)]$. The reason is that for all $\mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}$ we have:

$$\begin{array}{ll} & I[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}, Y/F(\mathbf{a_1}, \ldots, \mathbf{a_n})]\!] \models \psi \\ \text{iff} & I'[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}, Y/F(\mathbf{a_1}, \ldots, \mathbf{a_n})]\!] \models \psi \text{ since } f \text{ does not occur in } \psi \\ \text{iff} & I'[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!][\![Y/I'[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!](f(X_1, \ldots, X_n))]\!] \models \psi \\ \text{iff} & I'[\![X_1/\mathbf{a_1}, \ldots, X_n/\mathbf{a_n}]\!] \models \psi[Y/f(X_1, \ldots, X_n)] \end{array}$$

by the Substitution Lemma 2.2.3. Since the claim holds for all $\mathbf{a_1}, \ldots, \mathbf{a_n} \in \mathcal{A}$, we obtain $I' \models \forall X_1, \ldots, X_n\ \psi[Y/f(X_1, \ldots, X_n)]$. □

Example 3.1.7 *We consider the transformation of the formula*

$$\neg\exists X\ (\ \mathsf{married}(X, Y) \vee \neg\exists Y\ \mathsf{motherOf}(X, Y)\ )$$

from Example 3.1.3 into Skolem normal form. As in Example 3.1.3, the formula is first transformed into prenex normal form, which results in

$$\forall X\ \exists Z\ \neg(\mathsf{married}(X, Y) \vee \neg\mathsf{motherOf}(X, Z)).$$

Then the free variable Y is existentially quantified which yields

$$\exists Y\ \forall X\ \exists Z\ \neg(\mathsf{married}(X, Y) \vee \neg\mathsf{motherOf}(X, Z)).$$

To eliminate the existential quantifiers, one has to replace the outermost existentially quantified variable Y by a fresh constant a*. Here,* a *is 0-ary, since there is no universal quantifier before the existential quantifier of Y. This results in*

$$\forall X\, \exists Z\, \neg(\mathsf{married}(X, \mathsf{a}) \vee \neg\mathsf{motherOf}(X, Z)).$$

Finally, Z is replaced by $\mathsf{f}(X)$*, where* f *is a fresh 1-ary function symbol. This yields*

$$\forall X\, \neg(\mathsf{married}(X, \mathsf{a}) \vee \neg\mathsf{motherOf}(X, \mathsf{f}(X))).$$

3.2 Herbrand Structures

When analyzing unsatisfiability of a formula, the problem is that every interpretation resp. every structure needs to be considered and that the carrier can be a completely *arbitrary* set. In this section we show that for formulas in Skolem normal form, it is sufficient to consider only structures whose carrier consists of the ground terms and where the function symbols are interpreted as "themselves". These structures are called *Herbrand structures* (according to the logician *Jacques Herbrand*).

Definition 3.2.1 (Herbrand Structure) *Let (Σ, Δ) be a signature. A* Herbrand structure *(or "free" structure) has the form $(\mathcal{T}(\Sigma), \alpha)$, where for all $f \in \Sigma_n$ with $n \in \mathbb{N}$, we have $\alpha_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$. If a Herbrand structure is a model of a formula φ, then we call it a* Herbrand model *of φ.*

Note that for a Herbrand structure S, all ground terms t are interpreted as "themselves", i.e., we have $S(t) = t$, as can easily be shown by induction. Thus, for a Herbrand structure the carrier and the interpretation of the function symbols is fixed. Only the interpretation of the predicate symbols can be chosen freely. If only Herbrand structures instead of arbitrary structures are needed to analyze unsatisfiability, then the search space is already restricted considerably.

Example 3.2.2 *We again consider the signature (Σ, Δ) from Example 2.1.2. A Herbrand structure for this signature is for example $S = (\mathcal{T}(\Sigma), \alpha)$ with*

$$\begin{aligned}
\alpha_n &= n \text{ for all } n \in \mathbb{N} \\
\alpha_{\mathsf{monika}} &= \mathsf{monika} \\
\alpha_{\mathsf{karin}} &= \mathsf{karin} \\
&\vdots \\
\alpha_{\mathsf{date}}(t_1, t_2, t_3) &= \mathsf{date}(t_1, t_2, t_3) \text{ for all } t_1, t_2, t_3 \in \mathcal{T}(\Sigma) \\
\alpha_{\mathsf{female}} &= \{\mathsf{monika}, \mathsf{karin}, \ldots\} \\
\alpha_{\mathsf{male}} &= \{\mathsf{werner}, \mathsf{klaus}, \ldots\} \\
\alpha_{\mathsf{human}} &= \mathcal{T}(\Sigma) \\
\alpha_{\mathsf{born}} &= \{(\mathsf{monika}, \mathsf{date}(25, 7, 1972)), (\mathsf{werner}, \mathsf{date}(12, 7, 1969)), \ldots\} \\
&\vdots
\end{aligned}$$

This Herbrand structure is already a lot closer to the intuitively expected semantics of the original logic program.

The following theorem shows that we can restrict ourselves to Herbrand structures when analyzing satisfiability. To this end, as in Theorem 3.1.6 we have to transform the original formula into a satisfiability-equivalent formula in Skolem normal form.

Theorem 3.2.3 (Herbrand Structures are Sufficient for Unsatisfiability) *Let $\Phi \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ be a set of formulas in Skolem normal form. Then Φ is satisfiable iff it has a Herbrand model.*

Proof. The direction "$\Leftarrow$" is trivial, since every Herbrand model is also a model. We now show the other direction "$\Rightarrow$". Let $S = (\mathcal{A}, \alpha)$ be a model of Φ. (It suffices to consider structures instead of interpretations, since formulas in Skolem normal form are closed.) We show that then the Herbrand structure $S' = (\mathcal{T}(\Sigma), \alpha')$ is also a model of Φ. As for every Herbrand structure, we have $\alpha'_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for all $f \in \Sigma_n$ and all $n \in \mathbb{N}$. The interpretation of the predicate symbols is defined as follows. For $p \in \Delta_0$ let $\alpha'_p = \alpha_p$ and for all $p \in \Delta_n$ with $n \geq 1$ let

$$(t_1, \ldots, t_n) \in \alpha'_p \qquad \text{iff} \qquad (S(t_1), \ldots, S(t_n)) \in \alpha_p.$$

We show that for every formula φ in Skolem normal form, $S \models \varphi$ implies $S' \models \varphi$. Since φ is in Skolem normal form, it is of the form $\forall X_1, \ldots, X_n\ \psi$, where ψ is quantifier-free. To prove the above claim, we use induction on the number n of universally quantified variables of φ.

In the induction base we have $n = 0$, i.e., $\varphi = \psi$ is a quantifier-free formula without variables. Here, we even have "$S \models \varphi$ iff $S' \models \varphi$", as can be shown by structural induction on the form of the formula. For atomic formulas $p(t_1, \ldots, t_n)$ (i.e., in the induction base), by definition we have:

$$\begin{array}{lll} S \models p(t_1, \ldots, t_n) & \text{iff} & (S(t_1), \ldots, S(t_n)) \in \alpha_p \\ & \text{iff} & (t_1, \ldots, t_n) \in \alpha'_p \qquad \text{(by definition)} \\ & \text{iff} & (S'(t_1), \ldots, S'(t_n)) \in \alpha'_p \quad \text{(since } S'(t_i) = t_i\text{)} \\ & \text{iff} & S' \models p(t_1, \ldots, t_n) \end{array}$$

The proof for non-atomic quantifier-free formulas (i.e., the induction step) is trivial.

Now we consider the case $n > 0$, i.e., the induction step of the outer induction. Note that the formula $\forall X_1, \ldots, X_{n-1}\ \psi$ that results from omitting the quantifier "$\forall X_n$" is in general not a closed formula, since it could contain the free variable X_n. In the following, let $S[\![X_n/\mathbf{a}]\!]$ denote an interpretation which results from the structure S by adding the variable assignment $\beta[\![X_n/\mathbf{a}]\!]$ for an arbitrary variable assignment β. If $I = (\mathcal{A}, \alpha, \beta)$ then

$S[\![X_n/\mathbf{a}]\!]$ is an abbreviation for $I[\![X_n/\mathbf{a}]\!]$. We have:

$$
\begin{array}{lll}
& S \models \forall X_1, \ldots, X_n\ \psi & \\
\text{iff} & S[\![X_n/\mathbf{a}]\!] \models \forall X_1, \ldots, X_{n-1}\ \psi \text{ for all } \mathbf{a} \in \mathcal{A} & \\
\curvearrowright & S[\![X_n/S(t)]\!] \models \forall X_1, \ldots, X_{n-1}\ \psi \text{ for all } t \in \mathcal{T}(\Sigma) & \\
\text{iff} & S \models \forall X_1, \ldots, X_{n-1}\ \psi[X_n/t] \text{ for all } t \in \mathcal{T}(\Sigma) & \text{(Substitution Lemma 2.2.3)} \\
\curvearrowright & S' \models \forall X_1, \ldots, X_{n-1}\ \psi[X_n/t] \text{ for all } t \in \mathcal{T}(\Sigma) & \text{(induction hypothesis)} \\
\text{iff} & S'[\![X_n/S'(t)]\!] \models \forall X_1, \ldots, X_{n-1}\ \psi \text{ for all } t \in \mathcal{T}(\Sigma) & \text{(Substitution Lemma 2.2.3)} \\
\text{iff} & S'[\![X_n/t]\!] \models \forall X_1, \ldots, X_{n-1}\ \psi \text{ for all } t \in \mathcal{T}(\Sigma) & \text{(since } S'(t) = t \text{ for } t \in \mathcal{T}(\Sigma)\text{)} \\
\text{iff} & S' \models \forall X_1, \ldots, X_n\ \psi &
\end{array}
$$

□

Example 3.2.4 *The following example shows that the theorem above indeed only holds for formulas in Skolem normal form. The set of formulas* $\{\mathsf{p}(\mathsf{a}), \exists X\ \neg\mathsf{p}(X)\}$ *over the signature with* $\Sigma = \Sigma_0 = \{\mathsf{a}\}$ *and* $\Delta = \Delta_1 = \{\mathsf{p}\}$ *is satisfiable. For example, the structure* $S = (\{0,1\}, \alpha)$ *with* $\alpha_{\mathsf{a}} = 0$ *and* $\alpha_{\mathsf{p}} = \{0\}$ *is a model. But here it is important that there are objects in the carrier (as the object* 1 *in our example) that cannot be reached by an interpretation of a ground term (i.e., there is no ground term* t *with* $S(t) = 1$*). Indeed, this set of formulas does not have any Herbrand model. The reason is that the carrier of every Herbrand structure consists of the set* $\{\mathsf{a}\}$*.*

Now that we know that for proving the unsatisfiability of a formula φ in Skolem normal form we only need to consider Herbrand structures, we now reduce this task to the task of proving the unsatisfiability of an (infinite) set of formulas without variables. We will see that this corresponds to proving the unsatisfiability of an (infinite) set of *propositional logic* formulas. The idea is to instantiate all universally quantified variables in φ with all possible ground terms. In this way, we obtain the *Herbrand expansion* of φ.

Definition 3.2.5 (Herbrand Expansion of a Formula) *Let* $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ *be a formula in Skolem normal form, where* $\varphi = \forall X_1, \ldots, X_n\ \psi$ *for a quantifier-free formula* ψ*. The following set of formulas* $E(\varphi)$ *is called the* Herbrand expansion *of* φ*:*

$$E(\varphi) = \{\psi[X_1/t_1, \ldots, X_n/t_n] \mid t_1, \ldots, t_n \in \mathcal{T}(\Sigma)\}$$

So the Herbrand expansion of a formula $\forall X_1, \ldots, X_n\ \psi$ is the set of all *ground instances* of ψ.

Example 3.2.6 *We consider the formula* φ

$$\forall X\ \mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg\mathsf{motherOf}(X, \mathsf{susanne})$$

from Example 3.1.4, whose unsatisfiability has to be shown in order to prove that the query

```
?- motherOf(X,susanne).
```

holds for the logic program with the fact

```
motherOf(renate,susanne).
```

We get

$$E(\varphi) = \{ \begin{array}{l} \mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \mathsf{motherOf}(\mathsf{karin}, \mathsf{susanne}), \\ \mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}), \\ \mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg \mathsf{motherOf}(\mathsf{date}(\mathsf{karin}, \mathsf{werner}, 1982), \mathsf{susanne}), \\ \ldots \quad \}. \end{array}$$

In the example above, $E(\varphi)$ contains the formula $\mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg\mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne})$ which is obviously unsatisfiable. The following theorem states that this holds for every unsatisfiable formula φ in Skolem normal form. Thus, one can reduce the check for unsatisfiability to the problem of analyzing the unsatisfiability of a set of formulas without variables. This theorem is based on the preceding Theorem 3.2.3, which justifies the restriction to Herbrand models.

Theorem 3.2.7 (Satisfiability of the Herbrand Expansion) *Let φ be a formula in Skolem normal form. Then φ is satisfiable iff $E(\varphi)$ is satisfiable.*

Proof. Since φ is in Skolem normal form, it has the form $\forall X_1, \ldots, X_n\ \psi$, where ψ is quantifier-free. Now we have:

	$\forall X_1, \ldots, X_n\ \psi$ is satisfiable	
iff	there is a Herbrand structure S with $S \models \forall X_1, \ldots, X_n\ \psi$	(Theorem 3.2.3)
iff	there is a Herbrand structure S with $S[\![X_1/t_1, \ldots, X_n/t_n]\!] \models \psi$ for all $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$	
iff	there is a Herbrand structure S with $S \models \psi[X_1/t_1, \ldots, X_n/t_n]$ for all $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$	(Substitution Lemma 2.2.3)
iff	there is a Herbrand structure S with $S \models E(\varphi)$	
iff	$E(\varphi)$ is satisfiable	(Theorem 3.2.3)

□

Formulas of predicate logic without variables correspond to formulas of propositional logic. The reason is that every atomic subformula $p(t_1, \ldots, t_n)$ can be seen as a propositional variable that can either be true or false. By Theorems 3.2.3 and 3.2.7 it is sufficient to check if the set $E(\varphi)$ has a Herbrand model. Herbrand structures only differ in the interpretation of the predicate symbols. So for formulas without variables, the search for a Herbrand model just means that one searches for an assignment of the propositional variables $p(t_1, \ldots, t_n)$ with "true" or "false". Thus, the question of unsatisfiability of a formula is now reduced to the search of a satisfying assignment for an (in general infinite) set of propositional formulas.

Example 3.2.8 *We again consider the Herbrand expansion of the formula φ from Example 3.2.6. When replacing every atomic formula $p(t_1, \ldots, t_n)$ without variables by a propositional variable $V_{p(t_1,\ldots,t_n)}$, we obtain the following infinite set of propositional formulas from $E(\varphi)$:*

$$\begin{array}{ll}\{ & V_{\mathsf{motherOf(renate,susanne)}} \wedge \neg V_{\mathsf{motherOf(karin,susanne)}}, \\ & V_{\mathsf{motherOf(renate,susanne)}} \wedge \neg V_{\mathsf{motherOf(renate,susanne)}}, \\ & V_{\mathsf{motherOf(renate,susanne)}} \wedge \neg V_{\mathsf{motherOf(date(karin,werner,1982),susanne)}}, \\ & \ldots \quad \}.\end{array}$$

The formula φ and therefore $E(\varphi)$ is now satisfiable iff there is an assignment of the propositional variables $V_{p(t_1,\ldots,t_n)}$ with "true" or "false" which satisfies the set of propositional formulas above. Obviously there is no such assignment, because the formula

$$V_{\mathsf{motherOf(renate,susanne)}} \wedge \neg V_{\mathsf{motherOf(renate,susanne)}}$$

is neither satisfied for the assignment of $V_{\mathsf{motherOf(renate,susanne)}}$ with "true" nor with "false".

By the compactness theorem of propositional logic, an (infinite) set of propositional formulas is unsatisfiable iff it has a *finite* unsatisfiable subset (see e.g., [Gra11, Satz 1.15] and [Sch00, Kap. 1.4]). If $E(\varphi)$ is unsatisfiable, then there is already a finite unsatisfiable subset of $E(\varphi)$. Using this result, we can present the first procedure for checking unsatisfiability resp. for checking entailment. This procedure is a semi-decision algorithm, i.e., if the entailment holds (resp. if the corresponding formula is unsatisfiable), then the algorithm determines this and terminates.

This algorithm is called *Algorithm of Gilmore*. The input are formulas $\varphi_1, \ldots, \varphi_k, \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ and the goal is to find out whether $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ holds.

Algorithm of Gilmore

1. *Let ξ be the formula $\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi$. By Lemma 3.0.1, $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ holds iff ξ is unsatisfiable.*

2. *Transform ξ into Skolem normal form as in Theorems 3.1.2 and 3.1.6. The resulting formula ψ is satisfiability-equivalent to the original formula.*

3. *Choose a fair[1] enumeration $\{\psi_1, \psi_2, \ldots\} = E(\psi)$ of the Herbrand expansion of ψ and replace all atomic formulas by propositional variables. By Theorems 3.2.3 and 3.2.7, $E(\psi)$ is satisfiable iff ψ is satisfiable. By the compactness theorem of propositional logic, this is the case iff all finite subsets of $E(\psi)$ are satisfiable.*

4. *Check if ψ_1, $\psi_1 \wedge \psi_2$, $\psi_1 \wedge \psi_2 \wedge \psi_3$, ... are satisfiable (by testing all possible truth assignments of the occurring propositional variables). If one of those formulas is not satisfiable, stop and return "**true**".*

Thus, this algorithm terminates and returns the result "**true**" iff $\{\varphi_1, \ldots, \varphi_k\} \models \varphi$ holds. If this does not hold, then the algorithm does not terminate (provided that ψ contains variables and there are infinitely many ground terms). However, the drawback of this algorithm is that it is not goal-directed and therefore very inefficient.

[1] Here, "fairness" means that every formula from $E(\psi)$ appears at some point in the enumeration, i.e., for every formula $\psi' \in E(\psi)$ there is an i with $\psi_i = \psi'$.

Example 3.2.9 *To check*

$$\mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne}) \wedge \neg\exists\, X\ \mathsf{motherOf}(X, \mathsf{susanne}),$$

one needed to analyze the (propositional) satisfiability of the set of formulas from Example 3.2.8. To this end, it is now enough to consider its finite subsets. Choosing the enumeration that corresponds to the order in Example 3.2.8, then the unsatisfiability is already obvious in the second step (i.e., as soon as the subformula

$$V_{\mathsf{motherOf}(\mathsf{renate},\mathsf{susanne})} \wedge \neg V_{\mathsf{motherOf}(\mathsf{renate},\mathsf{susanne})}$$

is considered).

3.3 Ground Resolution

To make the check for unsatisfiability more efficient, in this section we introduce the proof method of *resolution*. We restrict ourselves to resolution in propositional logic in this section and extend it to resolution in predicate logic in Section 3.4. Propositional resolution is also called *ground resolution*, since here we only consider formulas without variables (i.e., formulas with *ground terms*).

To check unsatisfiability of a formula of the form $\forall\, X_1, \ldots, X_n\ \psi$ in Skolem normal form with the resolution calculus, the formula ψ first needs to be transformed into *conjunctive normal form (CNF)*. Such formulas will then be represented as *clause sets*.

Definition 3.3.1 (Conjunctive Normal Form, Literal, Clause) *A formula ψ is in* conjunctive normal form (CNF) *iff it is quantifier-free and has the following form.*

$$(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{m,n_m})$$

Here, the $L_{i,j}$ are literals*, i.e., they are atomic formulas or negated atomic formulas of the form $p(t_1, \ldots, t_k)$ or $\neg p(t_1, \ldots, t_k)$. For every literal L, its negation $\overline{L}$ is defined as follows:*

$$\overline{L} = \begin{cases} \neg A, & \text{if } L = A \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \\ A, & \text{if } L = \neg A \text{ with } A \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V}) \end{cases}$$

A set of literals is called a clause*. Every formula ψ in CNF can be represented by a corresponding* clause set

$$\mathcal{K}(\psi) = \{\{L_{1,1}, \ldots, L_{1,n_1}\}, \ldots, \{L_{m,1}, \ldots, L_{m,n_m}\}\}.$$

A clause represents the universally quantified disjunction *of its literals, and a clause set represents the* conjunction *of its clauses. Therefore, we also speak of* satisfiability *and* entailment *of clause sets. If not stated otherwise, in the following we only consider finite clause sets. The empty clause is denoted by $\square$ and is unsatisfiable by definition.*

Every quantifier-free formula can easily and automatically be transformed into CNF (resp. into its corresponding clause set).

Theorem 3.3.2 (Transformation to CNF) *For every quantifier-free formula ψ one can automatically construct a formula ψ' in conjunctive normal form such that ψ and ψ' are equivalent.*

Proof. First, we replace all subformulas $\psi_1 \leftrightarrow \psi_2$ in ψ by $\psi_1 \rightarrow \psi_2 \wedge \psi_2 \rightarrow \psi_1$. Then we replace all subformulas $\psi_1 \rightarrow \psi_2$ in ψ by $\neg\psi_1 \vee \psi_2$. The transformation of the remaining formula is done by the following algorithm CNF, whose termination and soundness is obvious. The input of the algorithm is an arbitrary quantifier-free formula ψ without the Boolean operators "$\leftrightarrow$" and "$\rightarrow$". As output, the algorithm returns an equivalent formula in CNF.

- If ψ is atomic, then return ψ.
- If $\psi = \psi_1 \wedge \psi_2$, then return $CNF(\psi_1) \wedge CNF(\psi_2)$.
- If $\psi = \psi_1 \vee \psi_2$, then compute $CNF(\psi_1) = \bigwedge_{i\in\{1,\ldots,m_1\}} \psi_i'$ and $CNF(\psi_2) = \bigwedge_{j\in\{1,\ldots,m_2\}} \psi_j''$. Here, ψ_i' and ψ_i'' are disjunctions of literals. By applying the distributive law, we get the formula $\bigwedge_{i\in\{1,\ldots,m_1\},\, j\in\{1,\ldots,m_2\}} \psi_i' \vee \psi_j''$ in CNF, which is returned.
- If $\psi = \neg\psi_1$, then compute $CNF(\psi_1) = \bigwedge_{i\in\{1,\ldots,m\}}(\bigvee_{j\in\{1,\ldots,n_i\}} L_{i,j})$. Multiple applications of De Morgan's law on the formula $\neg \bigwedge_{i\in\{1,\ldots,m\}}(\bigvee_{j\in\{1,\ldots,n_i\}} L_{i,j})$ yield $\bigvee_{i\in\{1,\ldots,m\}}(\bigwedge_{j\in\{1,\ldots,n_i\}} \overline{L_{i,j}})$. Then we proceed as in the previous case and use the distributive law. This results in the formula $\bigwedge_{j_1\in\{1,\ldots,n_1\},\ldots,j_m\in\{1,\ldots,n_m\}} \overline{L_{1,j_1}} \vee \ldots \vee \overline{L_{m,j_m}}$ in CNF, which is returned.

□

Example 3.3.3 *As an example we consider the following formula, where $\mathsf{p}, \mathsf{q}, \mathsf{r} \in \Delta_0$.*

$$\neg(\neg\mathsf{p} \wedge (\neg\mathsf{q} \vee \mathsf{r}))$$

By application of De Morgan's law, we get

$$\mathsf{p} \vee (\mathsf{q} \wedge \neg\mathsf{r}).$$

Finally, the distributive law yields the desired formula in CNF:

$$(\mathsf{p} \vee \mathsf{q}) \wedge (\mathsf{p} \vee \neg\mathsf{r})$$

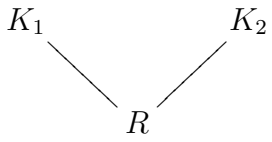
The basic idea of the algorithm of Gilmore was that unsatisfiability of a formula ψ in predicate logic can be reduced to the unsatisfiability of a set $E(\psi)$ of propositional formulas (resp. of formulas without variables). To check the unsatisfiability of $E(\psi)$, in the algorithm of Gilmore we searched for a finite unsatisfiable subset and then the unsatisfiability was proven by testing all truth assignments. However, proving unsatisfiability in propositional logic can be done much more goal-oriented by the *resolution calculus* if the quantifier-free part of the formula is in CNF. To this end, we define the notion of a *resolvent* for propositional (resp. variable-free) clauses.

Definition 3.3.4 (Propositional Resolution) *Let K_1 and K_2 be two clauses without variables. Then the clause R is a* resolvent *of K_1 and K_2 iff there is a literal $L \in K_1$ with $\overline{L} \in K_2$ and $R = (K_1 \backslash \{L\}) \cup (K_2 \backslash \{\overline{L}\})$. For a set of clauses $\mathcal{K}$, we define $Res(\mathcal{K}) = \mathcal{K} \cup \{R \mid R$ is a resolvent of two clauses from $\mathcal{K}\}$. Moreover, let $Res^0(\mathcal{K}) = \mathcal{K}$ and $Res^{n+1}(\mathcal{K}) = Res(Res^n(\mathcal{K}))$ for all $n \geq 0$. Finally, we define the set of clauses that can be derived by resolution from $\mathcal{K}$ as $Res^*(\mathcal{K}) = \bigcup_{n \geq 0} Res^n(\mathcal{K})$.*

Clearly, $\square \in Res^*(\mathcal{K})$ holds iff there is a sequence of clauses $K_1, \ldots, K_m$, such that $K_m = \square$ and such that for all $1 \leq i \leq m$, we have:

- $K_i \in \mathcal{K}$ or
- K_i is a resolvent of K_j and K_k for $j, k < i$

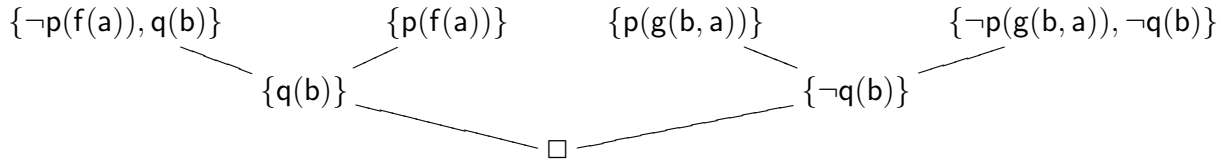
To denote resolution proofs, we often use the following diagram to depict explicitly that R is derived by resolution from K_1 and K_2.



Example 3.3.5 *Let $\Delta_1 = \{\mathsf{p}, \mathsf{q}\}$, $\Sigma_0 = \{\mathsf{a}, \mathsf{b}\}$, $\Sigma_1 = \{\mathsf{f}\}$, and $\Sigma_2 = \{\mathsf{g}\}$. We consider the set of clauses $\{K_1, K_2, K_3, K_4\}$ over this signature with*

$$\begin{array}{lcl} K_1 & = & \{\neg \mathsf{p}(\mathsf{f}(\mathsf{a})), \mathsf{q}(\mathsf{b})\} \\ K_2 & = & \{\mathsf{p}(\mathsf{f}(\mathsf{a}))\} \\ K_3 & = & \{\mathsf{p}(\mathsf{g}(\mathsf{b}, \mathsf{a}))\} \\ K_4 & = & \{\neg \mathsf{p}(\mathsf{g}(\mathsf{b}, \mathsf{a})), \neg \mathsf{q}(\mathsf{b})\} \end{array}$$

We have:



Theorem 3.3.7 shows the soundness and completeness of the resolution calculus in propositional logic. Thus, a set of clauses without variables is unsatisfiable iff the empty clause can be derived by repeated application of resolution steps. Thus, the set $\{K_1, K_2, K_3, K_4\}$ from Example 3.3.5 is unsatisfiable. To show the soundness and completeness of resolution in propositional logic, we first prove the following lemma that states that adding resolvents is an equivalence-preserving operation.

Lemma 3.3.6 (Propositional Resolution Lemma) *Let $\mathcal{K}$ be a set of clauses without variables. If $K_1, K_2 \in \mathcal{K}$ and R is a resolvent of K_1 and K_2, then $\mathcal{K}$ and $\mathcal{K} \cup \{R\}$ are equivalent.*

Proof. Every structure S that satisfies $\mathcal{K} \cup \{R\}$ also satisfies $\mathcal{K}$ (since a set of clauses represents the *conjunction* of its clauses). Therefore, $\mathcal{K} \cup \{R\} \models \mathcal{K}$ holds.

For the other direction, let S be a structure that satisfies $\mathcal{K}$. There is a literal $L \in K_1$ with $\overline{L} \in K_2$ and $R = (K_1 \setminus \{L\}) \cup (K_2 \setminus \{\overline{L}\})$. We assume that $S \not\models \mathcal{K} \cup \{R\}$ holds. Then $S \models \mathcal{K}$ implies $S \not\models R$. If $S \models L$, then $S \models K_2$ also implies $S \models K_2 \setminus \{\overline{L}\}$ and therefore $S \models R$. If $S \models \overline{L}$, then $S \models K_1$ also implies $S \models K_1 \setminus \{L\}$ and therefore $S \models R$. Thus $\mathcal{K} \models \mathcal{K} \cup \{R\}$ holds as well. □

Now we can prove the soundness and completeness of the propositional resolution calculus. Here, *soundness* means that we cannot prove any incorrect statements (i.e., that the set of clauses is indeed unsatisfiable if the empty clause can be derived by resolution). *Completeness* means that all true statements can be proven by resolution (i.e., that the empty clause can be derived by resolution from every unsatisfiable set of clauses).

Theorem 3.3.7 (Soundness and Completeness of Propositional Resolution) *Let $\mathcal{K}$ be a possibly infinite set of clauses without variables. Then $\mathcal{K}$ is unsatisfiable iff $\square \in Res^*(\mathcal{K})$.*

Proof. First we show the *soundness* (i.e., the direction "$\Leftarrow$"). By the Resolution Lemma 3.3.6, $\mathcal{K}$ and $Res(\mathcal{K})$ are equivalent. By induction on n, one can immediately show that then also $\mathcal{K}$ and $Res^n(\mathcal{K})$ are equivalent for all $n \in \mathbb{N}$. Clearly, $\square \in Res^*(\mathcal{K})$ means that there is an $n \in \mathbb{N}$ with $\square \in Res^n(\mathcal{K})$. Therefore $Res^n(\mathcal{K})$ is unsatisfiable. Since $\mathcal{K}$ and $Res^n(\mathcal{K})$ are equivalent, then $\mathcal{K}$ is also unsatisfiable.

Now we prove the *completeness* (i.e., the direction "$\Rightarrow$"). If $\mathcal{K}$ is unsatisfiable, then there is already a finite unsatisfiable subset $\mathcal{K}' \subseteq \mathcal{K}$ by the compactness theorem of propositional logic. Let n be the number of different atomic formulas in set $\mathcal{K}'$. We show $\square \in Res^*(\mathcal{K}')$ by induction on n.

In the induction base we consider the case $n = 0$. The only two clause sets without atomic formulas are $\varnothing$ and $\{\square\}$. Since $\varnothing$ is satisfiable (even valid), we must have $\mathcal{K}' = \{\square\}$ and therefore $\square \in Res^0(\mathcal{K}')$.

In the induction step $n > 0$, let A be an atomic formula that occurs in $\mathcal{K}'$. The set $\mathcal{K}^+$ results from $\mathcal{K}'$ by removing all clauses that contain A and by deleting $\neg A$ from all remaining clauses. Analogously, the set $\mathcal{K}^-$ results from $\mathcal{K}'$ by removing all clauses that contain $\neg A$ and by deleting A from all remaining clauses. So formally, $\mathcal{K}^+ = \{K \setminus \{\neg A\} \mid K \in \mathcal{K}', A \notin K\}$ and $\mathcal{K}^- = \{K \setminus \{A\} \mid K \in \mathcal{K}', \neg A \notin K\}$.

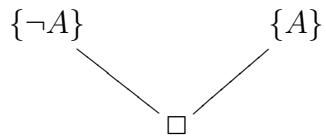
The unsatisfiability of $\mathcal{K}'$ also implies the unsatisfiability of the set $\mathcal{K}^+$. The reason is that if there were a structure S with $S \models \mathcal{K}^+$, then we could extend S to a structure S' with $S' \models A$ and would then have $S' \models \mathcal{K}'$. Analogously, $\mathcal{K}^-$ is unsatisfiable as well.

Therefore, the induction hypothesis implies $\square \in Res^*(\mathcal{K}^+)$ and $\square \in Res^*(\mathcal{K}^-)$. Note that $\square \in Res^*(\mathcal{K}^+)$ implies that there is a sequence of clauses $K_1, \ldots, K_m$ such that $K_m = \square$ and such that for all $1 \leq i \leq m$ we have:

- $K_i \in \mathcal{K}^+$ or
- K_i is a resolvent of K_j and K_k for $j, k < i$

If the required $K_i \in \mathcal{K}^+$ are already contained in $\mathcal{K}'$ (i.e., if the corresponding clauses from $\mathcal{K}'$ did not contain the literal $\neg A$), then this is also a derivation from $\mathcal{K}'$. Then $K_1, \ldots, K_m \in Res^*(\mathcal{K}')$ holds and therefore we have $\square \in Res^*(\mathcal{K}')$. Otherwise, by inserting $\neg A$ back in, we get a sequence of clauses $K'_1, \ldots, K'_m$ which proves that $\{\neg A\} \in Res^*(\mathcal{K}')$. The reason is the following: If C_i is a resolvent of C_j and C_k, then $C_i \cup \{\neg A\}$ is a resolvent of $C_j \cup \{\neg A\}$ and C_k:

Analogously, $\square \in Res^*(\mathcal{K}^-)$ implies that $\square \in Res^*(\mathcal{K}')$ or $\{A\} \in Res^*(\mathcal{K}')$. But $\{\neg A\}, \{A\} \in Res^*(\mathcal{K}')$ again implies $\square \in Res^*(\mathcal{K}')$:



□

Now the algorithm of Gilmore can be improved to the *ground resolution algorithm*. As in Steps 1 and 2 of the algorithm of Gilmore, let us first transform the problem of entailment into the problem of unsatisfiability of a formula $\forall X_1, \ldots, X_n\ \psi$ in Skolem normal form. Then we proceed as follows:

Ground Resolution Algorithm

3. *Transform ψ as in Theorem 3.3.2 into CNF resp. into the corresponding set of clauses $\mathcal{K}(\psi)$.*

4. *Choose a fair enumeration $\{K_1, K_2, \ldots\}$ of all ground instances of clauses from $\mathcal{K}(\psi)$. This corresponds to the Herbrand Expansion of $\mathcal{K}(\psi)$, i.e., the set of clauses $\{\sigma(K) \mid K \in \mathcal{K}(\psi), \sigma$ is ground substitution, i.e., $\mathcal{V}(\sigma(K)) = \varnothing\}$. Here, the application of a substitution on a clause means that the substitution is applied to all literals of the clause.*

5. *Compute $Res^*(\{K_1, K_2\})$, $Res^*(\{K_1, K_2, K_3\})$, ... If one of these sets contains the empty clause $\square$, stop and return "**true**".*

Example 3.3.8 *As an example we consider the signature from Example 3.3.5. We want to show the unsatisfiability of the following formula in Skolem normal form, whose quantifier-free part ψ is already in CNF:*

$$\forall X, Y\ (\neg \mathsf{p}(X) \vee \neg \mathsf{p}(\mathsf{f}(\mathsf{a})) \vee \mathsf{q}(Y)) \wedge \mathsf{p}(Y) \wedge (\neg \mathsf{p}(\mathsf{g}(\mathsf{b}, X)) \vee \neg \mathsf{q}(\mathsf{b}))$$

We obtain the following set of clauses $\mathcal{K}(\psi)$:

$$\{\ \{\neg \mathsf{p}(X),\ \neg \mathsf{p}(\mathsf{f}(\mathsf{a})),\ \mathsf{q}(Y)\},\ \{\mathsf{p}(Y)\},\ \{\neg \mathsf{p}(\mathsf{g}(\mathsf{b}, X)),\ \neg \mathsf{q}(\mathsf{b})\}\ \}$$

Now we choose an enumeration $\{K_1, K_2, K_3, K_4, \ldots\}$ of the ground instances of the three clauses, where $K_1, \ldots, K_4$ are the variable-free clauses from Example 3.3.5. As shown in Example 3.3.5, we can derive the empty clause and thus, also prove the unsatisfiability of the original formula. Note that here, both K_2 and K_3 are different instances of the same clause $\{\mathsf{p}(Y)\}$. So this can be necessary in order to prove unsatisfiability. Note also that by creating the instances, several literals of a clause can be made the same. So K_1 is an instance of the clause $\{\neg \mathsf{p}(X),\ \neg \mathsf{p}(\mathsf{f}(\mathsf{a})),\ \mathsf{q}(Y)\}$, but K_1 only contains two literals, since by the substitution of X by $\mathsf{f}(\mathsf{a})$, the first two literals of the clause become equal.

The following theorem shows the soundness and completeness of the ground resolution algorithm. So this algorithm terminates and returns the result "**true**" iff $\forall\, X_1, \ldots, X_n\ \psi$ is unsatisfiable. If this does not hold, the algorithm does not terminate (provided that the set of ground instances is infinite).

Theorem 3.3.9 (Soundness and Completeness of Ground Resolution)

(a) If a set of clauses $\mathcal{K}$ is unsatisfiable, then there is a finite set of ground instances of clauses in $\mathcal{K}$ (i.e., a finite subset of $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma$ is ground substitution$\}$), which is also unsatisfiable.

(b) Let $\forall\, X_1, \ldots, X_n\ \psi$ be a formula in Skolem normal form, where ψ is in CNF. Then $\forall\, X_1, \ldots, X_n\ \psi$ is unsatisfiable iff there is a sequence of clauses $K_1, \ldots, K_m$ such that $K_m = \square$ and such that for all $1 \leq i \leq m$ we have:

- *K_i is ground instance of a clause in $\mathcal{K}(\psi)$ or*
- *K_i is a resolvent of K_j and K_k for $j, k < i$*

Proof.

(a) Let $\mathcal{K} = \{K_1, \ldots, K_r\}$, let ψ_i be the formula that results from the non-universally quantified disjunction of the literals in K_i (for all $1 \leq i \leq r$), let $\psi = \psi_1 \wedge \ldots \wedge \psi_r$, and let $X_1, \ldots, X_n$ be the variables in ψ. Then $\mathcal{K}$ corresponds to the formula $\forall\, X_1, \ldots, X_n\ \psi$ in Skolem normal form, where ψ is in CNF (and $\mathcal{K} = \mathcal{K}(\psi)$ holds). We have:

$\mathcal{K}$ is unsatisfiable
iff $\forall\, X_1, \ldots, X_n\ \psi$ is unsatisfiable
iff $E(\forall X_1, \ldots, X_n\ \psi) = \{\sigma(\psi) \mid \sigma$ is ground substitution$\}$ is unsatisfiable (Theorem 3.2.7)
iff $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma$ is ground substitution$\}$ is unsatisfiable

By the compactness theorem of propositional logic, this is equivalent to the existence of a finite unsatisfiable subset of $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma$ is ground substitution$\}$.

(b) As in Part (a), $\forall X_1, \ldots, X_n\ \psi$ is unsatisfiable iff a finite subset of $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma \text{ is ground substitution}\}$ is unsatisfiable. By Theorem 3.3.7, this is equivalent to a derivation of the empty clause $\square$ by resolution from a finite subset of $\{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma \text{ is ground substitution}\}$. $\square$

However, the disadvantage of the algorithm is that the search for suitable ground instances is not yet goal-oriented and therefore, we have to try all ground instances in a very inefficient way. For example, decisions for some of the ground substitutions have to be made in a "prospective manner" in order to enable later resolution steps. This suggests a modification in which the substitutions are chosen as "general" as possible, i.e., one only performs those substitutions that are necessary for the next resolution step. To this end, we now permit substitutions with arbitrary terms instead of only ground substitutions. This leads to the *resolution calculus in predicate logic*, which will be introduced in the next section.

3.4 Resolution in Predicate Logic and Unification

The following examples illustrates the idea of the "general" substitutions mentioned above.

Example 3.4.1 *We consider the following set of clauses, where* $\mathsf{p}, \mathsf{q} \in \Delta_1$, $\mathsf{f} \in \Sigma_1$, *and* $\mathsf{a} \in \Sigma_0$.

$$\{\ \{\mathsf{p}(X), \neg\mathsf{q}(X)\},\ \{\neg\mathsf{p}(\mathsf{f}(Y))\},\ \{\mathsf{q}(\mathsf{f}(\mathsf{a}))\}\ \}$$

To resolve the first two clauses, we now use the substitution $\{X/\mathsf{f}(Y)\}$. *After applying this substitution we get the literals* $\mathsf{p}(X)[X/\mathsf{f}(Y)] = \mathsf{p}(\mathsf{f}(Y))$ *and* $\neg\mathsf{p}(\mathsf{f}(Y))[X/\mathsf{f}(Y)] = \neg\mathsf{p}(\mathsf{f}(Y))$ *which are complementary to each other. The substitution* $\{X/\mathsf{f}(Y)\}$ *is therefore a* unifier *of* $\{\{\mathsf{p}(X), \mathsf{p}(\mathsf{f}(Y))\}$. *The resulting resolvent is the clause* $\{\neg\mathsf{q}(X)[X/\mathsf{f}(Y)]\} = \{\neg\mathsf{q}(\mathsf{f}(Y))\}$ *with the remaining second literal of the first clause.*

The advantage is that the substitution $\{X/\mathsf{f}(Y)\}$ *does not yet determine how to instantiate* Y *afterwards. It turns out that we have to use the substitution* $\{Y/\mathsf{a}\}$ *afterwards to finally derive the empty clause. But we do not have to recognize this from the beginning (i.e., we do not have to use the substitution* $\{X/\mathsf{f}(\mathsf{a})\}$ *right away), but we only use substitutions like* $\{X/\mathsf{f}(Y)\}$ *which are as general as possible. In other words, in the first step of the resolution we choose the* most general *unifier of* $\{\{\mathsf{p}(X), \mathsf{p}(\mathsf{f}(Y))\}$ *and not the less general unifier* $\{X/\mathsf{f}(\mathsf{a})\}$.

The following definition formally introduces the concept of unification.

Definition 3.4.2 (Unification) *A clause* $K = \{L_1, \ldots, L_n\}$ *is* unifiable *iff there is a substitution* σ *with* $\sigma(L_1) = \ldots = \sigma(L_n)$ *(i.e.,* $|\sigma(K)| = 1$*). Such a substitution is called a* unifier *of* K. *A unifier* σ *is a* most general *unifier (mgu) iff for every unifier* σ' *there is a substitution* δ *such that* $\sigma'(X) = \delta(\sigma(X))$ *for all* $X \in \mathcal{V}$.

If a clause is unifiable, then it also has a most general unifier that is unique up to variable renaming[2] (see, e.g., [Gie11]).

[2] A substitution σ is a *variable renaming* iff it is injective and we have $\sigma(X) \in \mathcal{V}$ for all $X \in \mathcal{V}$.

For any clause, it is decidable if it is unifiable. The following unification algorithm is due to *J. Robinson* [Rob65]. Its input is a clause $K = \{L_1, \ldots, L_n\}$ that has to be unified.

Unification Algorithm

1. *Let $\sigma = \varnothing$ be the empty (or "identity") substitution.*
2. *If $|\sigma(K)| = 1$, then stop and return σ as mgu of K.*
3. *Otherwise, search all $\sigma(L_i)$ in parallel from left to right until the corresponding symbols in two literals are different.*
4. *If none of the symbols is a variable, then stop with* clash failure.
5. *Otherwise, let X be the variable and let t be the corresponding subterm in the other literal (here, t can also be a variable). If X occurs in t, then stop with* occur failure. *(This check is called* occur check.*)*
6. *Otherwise set $\sigma = \{X/t\} \circ \sigma$ and go back to Step 2.*

Here, $\sigma_1 \circ \sigma_2$ is the composition of substitutions. Thus, we have $(\sigma_1 \circ \sigma_2)(X) = \sigma_1(\sigma_2(X))$.

Example 3.4.3 *Let $\mathsf{q} \in \Delta_1$, $\mathsf{p} \in \Delta_2$, $\mathsf{f}, \mathsf{g} \in \Sigma_2$, $\mathsf{h} \in \Sigma_1$, and $\mathsf{a} \in \Sigma_0$.*

As an example of a clash failure, we consider the clause

$$\{\ \mathsf{q}(\mathsf{f}(X, Y)),\ \mathsf{q}(\mathsf{g}(X, Y))\ \}.$$

On the first position where the two literals differ, we have the function symbol f in the first literal and the function symbol g in the second literal. There is no instantiation of the variables X and Y which can make these two literals equal because instantiations cannot change these different function symbols. Thus, this clause is not unifiable.

As an example of an occur failure we consider the following clause:

$$\{\ \mathsf{q}(X),\ \mathsf{q}(\mathsf{h}(X))\ \}.$$

On the first position where the two literals differ, we have the variable X in the first literal and the term $\mathsf{h}(X)$ in the second literal, which contains the variable X. There is no instantiation which makes these two terms (resp. these two literals) equal. Hence, this clause is not unifiable either.

Finally, we apply the unification algorithm to the clause consisting of the following two literals:

$$\begin{array}{l} \neg\mathsf{p}(\mathsf{f}(Z, \mathsf{g}(\mathsf{a}, Y)), \mathsf{h}(Z)) \\ \neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U, V), W), \mathsf{h}(\mathsf{f}(\mathsf{a}, Y))) \\ \qquad\ \uparrow \end{array}$$

The arrow indicates the first position where the two literals differ. This results in $\sigma = \{Z/\mathsf{f}(U, V)\}$. Now we apply the already found partial unifier σ to the literals:

$$\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(U,V)))$$
$$\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),W),\mathsf{h}(\mathsf{f}(\mathsf{a},Y)))$$
$$\uparrow$$

This yields $\sigma = \{W/\mathsf{g}(\mathsf{a},Y)\} \circ \sigma = \{Z/\mathsf{f}(U,V), W/\mathsf{g}(\mathsf{a},Y)\}$. *By application of* σ *we get:*

$$\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(U,V)))$$
$$\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(U,V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(\mathsf{a},Y)))$$
$$\uparrow$$

This results in $\sigma = \{U/\mathsf{a}\} \circ \sigma = \{Z/\mathsf{f}(\mathsf{a},V), W/\mathsf{g}(\mathsf{a},Y), U/\mathsf{a}\}$. *The application of* σ *yields:*

$$\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(\mathsf{a},V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(\mathsf{a},V)))$$
$$\neg\mathsf{p}(\mathsf{f}(\mathsf{f}(\mathsf{a},V),\mathsf{g}(\mathsf{a},Y)),\mathsf{h}(\mathsf{f}(\mathsf{a},Y)))$$
$$\uparrow$$

Finally, we get $\sigma = \{Y/V\} \circ \sigma = \{Z/\mathsf{f}(\mathsf{a},V), W/\mathsf{g}(\mathsf{a},V), U/\mathsf{a}, Y/V\}$. *Afterwards both instantiated literals are the same, i.e., this is the desired mgu.*

The following theorem shows the termination and soundness of the unification algorithm. (In [Gie11] there is a more formal version of the unification algorithm as a collection of four transformation rules, which also provides an exact formalization of the "parallel searching of literals from left to right". There, termination and soundness of this algorithm is also formally proven).

Theorem 3.4.4 (Termination and Soundness of the Unification Algorithm) *The unification algorithm terminates for every clause* $K \neq \square$ *and it is sound, i.e., it computes an mgu for the clause* K *iff* K *is unifiable.*

Proof. The algorithm terminates, since in each iteration of the loop from Step 2 – 6, the number of variables in $\sigma(K)$ decreases by 1.

If the algorithm terminates successfully and returns a substitution σ, σ is obviously a unifier of K, since $|\sigma(K)| = 1$. So if the clause K is not unifiable, the algorithm has to abort with a clash or occur failure.

We still have to show that for every unifiable clause, the algorithm indeed finds a unifier and that this unifier is then also a *most general* unifier.

Let $m \geq 0$ be the number of loop iterations that occur for the input clause K. For all $0 \leq i \leq m$, let σ_i be the value of σ after the i-th loop iteration. We show the following claim for all $0 \leq i \leq m$:

$$\text{For every unifier } \sigma' \text{ of } K, \text{ we have } \sigma' = \sigma' \circ \sigma_i. \tag{3.1}$$

Claim (3.1) implies that it is not possible to terminate with clash or occur failure. The reason is that if the $(m+1)$-th loop were aborted due to such a failure, then $\sigma_m(K)$ could not be unified. But since K is unifiable, K has a unifier $\sigma' = \sigma' \circ \sigma_m$. Therefore, σ' is also a unifier of $\sigma_m(K)$.

As the loop is only iterated m times, we must have $|\sigma_m(K)| = 1$, i.e., σ_m is a unifier of K. Since for each unifier σ' of K, there is a substitution $\delta = \sigma'$ with $\sigma' = \delta \circ \sigma_m$, σ_m is also a mgu of K.

We now prove the claim (3.1) by induction on i. In the induction base $i = 0$, $\sigma_0 = \varnothing$ is the identity. Therefore, we have $\sigma' = \sigma' \circ \sigma_0$ for all σ'.

In the induction step $i > 0$, in the i-th loop iteration a variable X is found in one literal and a term t is found in the other literal. The result is $\sigma_i = \{X/t\} \circ \sigma_{i-1}$. According to the induction hypothesis $\sigma' = \sigma' \circ \sigma_{i-1}$ holds for each unifier σ' of K. Thus, we have:

$$\begin{array}{lll} & \sigma' \circ \sigma_i & \\ = & \sigma' \circ \{X/t\} \circ \sigma_{i-1} & \text{(def. of } \sigma_i\text{)} \\ = & \sigma' \circ \sigma_{i-1} & \text{(since } \sigma' \circ \{X/t\} = \sigma'\text{)} \end{array}$$

In order to show $\sigma' \circ \{X/t\} = \sigma'$, first note that the two substitutions are obviously identical on all variables $Y \neq X$. For the variable X, the result is $(\sigma' \circ \{X/t\})(X) = \sigma'(t) = \sigma'(X)$, since σ' is a unifier of $\sigma_{i-1}(K)$ (because $|\sigma'(K)| = |\sigma'(\sigma_{i-1}(K))| = 1$) and each unifier of $\sigma_{i-1}(K)$ must also unify the terms X and t. □

With the concept of unification, we can now define resolution in predicate logic.

Definition 3.4.5 (Resolution in Predicate Logic) *Let K_1 and K_2 be clauses. Then the clause R is a* resolvent *of K_1 and K_2 iff the following three conditions hold:*

- *There exist variable renamings ν_1 and ν_2 such that $\nu_1(K_1)$ and $\nu_2(K_2)$ have no common variables.*
- *There are literals $L_1, \ldots, L_m \in \nu_1(K_1)$ and literals $L'_1, \ldots, L'_n \in \nu_2(K_2)$ with $n, m \geq 1$ such that $\{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$ is unifiable with a mgu σ.*
- $R = \sigma((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L'_1, \ldots, L'_n\}))$

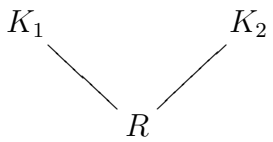
For a clause set $\mathcal{K}$, as in propositional resolution in Def. 3.3.4), we define:

$$\begin{array}{rcl} Res(\mathcal{K}) & = & \mathcal{K} \cup \{R \mid R \text{ is resolvent of two clauses from } \mathcal{K}\} \\ Res^0(\mathcal{K}) & = & \mathcal{K} \\ Res^{n+1}(\mathcal{K}) & = & Res(Res^n(\mathcal{K})) \text{ for all } n \geq 0 \\ Res^*(\mathcal{K}) & = & \bigcup_{n \geq 0} Res^n(\mathcal{K}) \end{array}$$

Clearly, propositional resolution is a special case of resolution in predicate logic, since for clauses without variables, the definition is the same as Def. 3.3.4. Similar to propositional resolution, $\square \in Res^*(\mathcal{K})$ holds iff there is a sequence of clauses $K_1, \ldots, K_m$ such that $K_m = \square$ and such that for all $1 \leq i \leq m$, we have:

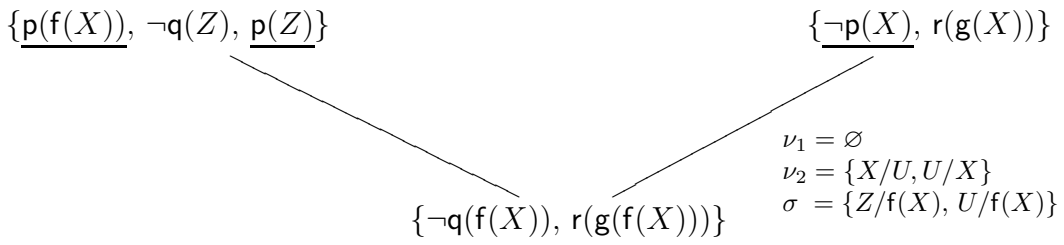
- $K_i \in \mathcal{K}$ or
- K_i is a resolvent of K_j and K_k for $j, k < i$

To depict resolution proofs, we again use the following diagram to show explicitly that R is derived by resolution from K_1 and K_2.



A more detailed representation is possible by underlining the eliminated literals and specifying the renamings and the unifier.

Example 3.4.6 *As an example, we consider the following resolution step, where* $\mathsf{p}, \mathsf{q}, \mathsf{r} \in \Delta_1$ *and* $\mathsf{f}, \mathsf{g} \in \Sigma_1$.



For the soundness proof of resolution in predicate logic resolution, we extend the Resolution Lemma 3.3.6 from propositional logic to predicate logic.

Lemma 3.4.7 (Resolution Lemma in Predicate Logic) *Let $\mathcal{K}$ be a set of clauses. If $K_1, K_2 \in \mathcal{K}$ and R is resolvent of K_1 and K_2, then $\mathcal{K}$ and $\mathcal{K} \cup \{R\}$ are equivalent.*

Proof. As in propositional logic, $S \models \mathcal{K} \cup \{R\}$ obviously implies $S \models \mathcal{K}$ for all structures S. Here, it is enough to consider *structures* instead of *interpretations*, since every clause represents the *universally quantified* disjunction of its literals and every set of clauses represents the conjunction of its clauses. Therefore $\mathcal{K} \cup \{R\} \models \mathcal{K}$ holds.

For the other direction, let S now be a structure that satisfies $\mathcal{K}$. We have

$$R = \sigma((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L'_1, \ldots, L'_n\})).$$

Here, ν_1 and ν_2 are variable renamings such that $\nu_1(K_1)$ and $\nu_2(K_2)$ do not contain common variables. Moreover, $L_1, \ldots, L_m \in \nu_1(K_1)$ and $L'_1, \ldots, L'_n \in \nu_2(K_2)$ with $n, m \geq 1$, such that $\{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$ are unifiable with the mgu σ. Thus, we have $\sigma(L_1) = \ldots = \sigma(L_m) = L$ and $\sigma(L'_1) = \ldots = \sigma(L'_n) = \overline{L}$ for some literal L.

We assume that $S \not\models \mathcal{K} \cup \{R\}$ holds. Then $S \models \mathcal{K}$ implies $S \not\models R$. Let

$$\nu_1(K_1) = \{L_1, \ldots, L_m, L_{m+1}, \ldots, L_p\} \qquad \text{and} \qquad \nu_2(K_2) = \{L'_1, \ldots, L'_n, L'_{n+1}, \ldots, L'_q\}$$

with $p \geq m$ and $q \geq n$. Then R is obtained by universally quantifying the formula

$$\sigma(L_{m+1} \vee \ldots \vee L_p \vee L'_{n+1} \vee \ldots \vee L'_q).$$

Let $S = (\mathcal{A}, \alpha)$. So there is an interpretation $I = (\mathcal{A}, \alpha, \beta)$ with

$$I \not\models \sigma(L_{m+1} \vee \ldots \vee L_p \vee L'_{n+1} \vee \ldots \vee L'_q).$$

Let $\sigma = \{X_1/t_1, \ldots, X_k/t_k\}$ and let I' be the interpretation $I[\![X_1/I(t_1), \ldots, X_k/I(t_k)]\!]$. By the Substitution Lemma 2.2.3 we then obtain

$$I' \not\models L_{m+1} \vee \ldots \vee L_p \vee L'_{n+1} \vee \ldots \vee L'_q. \tag{3.2}$$

But since $S \models K_1$ and $S \models K_2$ hold, $S \models \nu_1(K_1)$ and $S \models \nu_2(K_2)$ follow as well, and therefore

$$I' \models L_1 \vee \ldots \vee L_m \vee L_{m+1} \vee \ldots \vee L_p \qquad \text{and} \qquad I' \models L'_1 \vee \ldots \vee L'_n \vee L'_{n+1} \vee \ldots \vee L'_q.$$

With (3.2), we have

$$I' \models L_1 \vee \ldots \vee L_m \qquad \text{and} \qquad I' \models L'_1 \vee \ldots \vee L'_n$$

and with the Substitution Lemma 2.2.3 we have

$$I \models \sigma(L_1 \vee \ldots \vee L_m) \qquad \text{and} \qquad I \models \sigma(L'_1 \vee \ldots \vee L'_n).$$

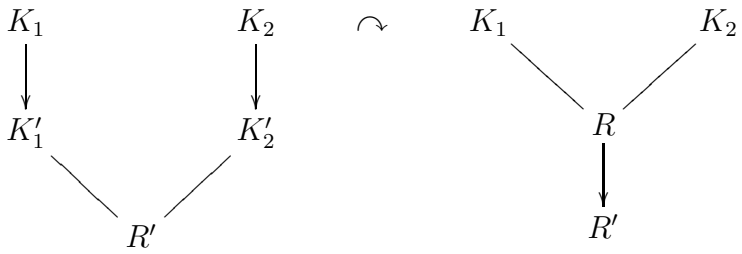
We thus obtain the following contradiction:

$$I \models L \qquad \text{and} \qquad I \models \overline{L}.$$

□

The following lemma is needed to reduce the completeness of resolution in predicate logic to the completeness of resolution in propositional logic. It shows that the (propositional) resolution step of two ground instances of clauses can be "lifted" to a resolution step of the two original clauses in predicate logic.

Lemma 3.4.8 (Lifting Lemma) *Let K_1 and K_2 be two clauses with ground instances K'_1 and K'_2. If R' is a resolvent of K'_1 and K'_2, then there is a resolvent R of K_1 and K_2, such that R' is a ground instance of R. The following diagram illustrates the Lifting Lemma, where arrows indicate ground instances.*



Proof. Let ν_1 and ν_2 be variable renamings such that $\nu_1(K_1)$ and $\nu_2(K_2)$ have no common variables. Since K'_i is a ground instance of K_i, it is also a ground instance of $\nu_i(K_i)$. Since $\nu_1(K_1)$ and $\nu_2(K_2)$ also have no common variables, one can choose a *common* ground substitution σ for $\nu_1(K_1)$ and $\nu_2(K_2)$ such that $\sigma(\nu_1(K_1)) = K'_1$ and $\sigma(\nu_2(K_2)) = K'_2$.

Since R' is a resolvent of K'_1 and K'_2, there is a literal $L \in K'_1$ with $\overline{L} \in K'_2$, such that $R' = (K'_1 \setminus \{L\}) \cup (K'_2 \setminus \{\overline{L}\})$.

We now select all literals from $\nu_1(K_1)$ resp. $\nu_2(K_2)$ that are mapped to L resp. $\overline{L}$ by σ. More precisely, let $L_1, \ldots, L_m \in \nu_1(K_1)$ and $L'_1, \ldots, L'_n \in \nu_2(K_2)$ be *all* literals from the respective clauses with $L = \sigma(L_1) = \ldots = \sigma(L_m)$ and $\overline{L} = \sigma(L'_1) = \ldots = \sigma(L'_n)$. Here $m, n \geq 1$ holds, since $L \in \sigma(\nu_1(K_1))$ and $\overline{L} \in \sigma(\nu_2(K_2))$.

Since σ is a unifier of $\{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$, there also exists an mgu σ'. Therefore K_1 and K_2 have the resolvent $R = \sigma'((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L'_1, \ldots, L'_n\}))$.

Since σ is a unifier of $\{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$ and σ' is the mgu, there exists a substitution δ with $\sigma = \delta \circ \sigma'$. Here, δ is a ground substitution, because σ is also a ground

substitution. Now we can show that the resolvent R' of K_1' and K_2' is also a ground instance of the resolvent R of K_1 and K_2:

$$\begin{array}{rcl} R' & = & (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\}) \\ & = & (\sigma(\nu_1(K_1)) \setminus \{L\}) \cup (\sigma(\nu_2(K_2)) \setminus \{\overline{L}\}) \\ & = & \sigma((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \ldots, L_n'\})) \\ & = & \delta(\sigma'((\nu_1(K_1) \setminus \{L_1, \ldots, L_m\}) \cup (\nu_2(K_2) \setminus \{L_1', \ldots, L_n'\}))) \\ & = & \delta(R) \end{array}$$

The step from the second to the third line is correct, since $L_1, \ldots, L_m$ resp. $L_1', \ldots, L_n'$ are *all* literals in $\nu_1(K_1)$ resp. $\nu_2(K_2)$ that are mapped to L resp. $\overline{L}$. □

Example 3.4.9 *The following example illustrates the Lifting Lemma. We consider the clauses* $K_1 = \{\mathsf{p}(\mathsf{f}(X)), \neg\mathsf{q}(Z), \mathsf{p}(Z)\}$ *and* $K_2 = \{\neg\mathsf{p}(X), \mathsf{r}(\mathsf{g}(X))\}$ *from Example 3.4.6. By instantiation with the ground substitution* $\{X/\mathsf{a}, Z/\mathsf{f}(\mathsf{a})\}$ *one obtains the ground instance* $K_1' = \{\mathsf{p}(\mathsf{f}(\mathsf{a}))\, \neg\mathsf{q}(\mathsf{f}(\mathsf{a}))\}$ *and instantiation with* $\{X/\mathsf{f}(\mathsf{a})\}$ *results in* $K_2' = \{\neg\mathsf{p}(\mathsf{f}(\mathsf{a})), \mathsf{r}(\mathsf{g}(\mathsf{f}(\mathsf{a})))\}$. *Here, we have* $\mathsf{a} \in \Sigma_0$. *The resolution of* K_1' *and* K_2' *results in* $R' = \{\neg\mathsf{q}(\mathsf{f}(\mathsf{a})), \mathsf{r}(\mathsf{g}(\mathsf{f}(\mathsf{a})))\}$. *One can see that this resolvent is an instance of the resolvent* $R = \{\neg\mathsf{q}(\mathsf{f}(X)), \mathsf{r}(\mathsf{g}(\mathsf{f}(X))\}$ *of* K_1 *and* K_2 *from Example 3.4.6.*

Now we can prove the soundness and completeness of resolution in predicate logic.

Theorem 3.4.10 (Soundness and Completeness of Resolution in Predicate Logic) *Let* $\mathcal{K}$ *be a (finite) set of clauses. Then* $\mathcal{K}$ *is unsatisfiable iff* $\square \in Res^*(\mathcal{K})$.

Proof. The *soundness* (i.e., the direction "⇐") follows from the Resolution Lemma in Predicate Logic (Lemma 3.4.7) analogously to the soundness proof for propositional resolution (Theorem 3.3.7).

Now we prove the *completeness* (i.e., the direction "⇒"). Since $\mathcal{K}$ is unsatisfiable, by Theorem 3.3.9 (a) there is a finite unsatisfiable set of ground instances of clauses from $\mathcal{K}$. By the completeness of the propositional resolution (Theorem 3.3.7), there is a sequence of clauses $K_1', \ldots, K_m'$ such that $K_m' = \square$ and such that for all $1 \leq i \leq m$ we have:

- K_i' is ground instance of a clause $K \in \mathcal{K}$ or
- K_i' is a resolvent of K_j' and K_k' for $j, k < i$

With the Lifting Lemma 3.4.8, we now generate a sequence of clauses $K_1, \ldots, K_m$, where K_i' is a ground instance of K_i and where $K_i \in Res^*(\mathcal{K})$. Here, we proceed as follows:

- If K_i' is a ground instance of a clause $K \in \mathcal{K}$, then we choose $K_i = K$.
- Otherwise, K_i' is a resolvent of K_j' and K_k' with $j, k < i$. There are already clauses K_j and K_k with $K_j, K_k \in Res^*(\mathcal{K})$ such that K_j' and K_k' are ground instances of K_j and K_k. By the Lifting Lemma 3.4.8, then there is a resolvent of K_j and K_k such that K_i' is a ground instance of this resolvent. We choose this resolvent as K_i. Since $K_j, K_k \in Res^*(\mathcal{K})$ holds, we obtain $K_i \in Res^*(\mathcal{K})$.

This implies that $K'_m = \square$ is a ground instance of K_m, i.e., $K_m = \square$. It also implies that $K_m = \square \in Res^*(\mathcal{K})$. $\square$

So we now obtain the following algorithm to solve the entailment problem. As in Steps 1 and 2 of the algorithm of Gilmore, we transform the entailment problem into the problem of showing the unsatisfiability of a formula $\forall X_1, \ldots, X_n\ \psi$ in Skolem normal form. Then we proceed as follows:

Resolution Algorithm

3. *Transform ψ as in Theorem 3.3.2 into CNF resp. into the corresponding set of clauses $\mathcal{K}(\psi)$.*

4. *Compute $Res^*(\mathcal{K}(\psi))$. If the empty clause was found, stop and return "**true**". If $Res^*(\mathcal{K}(\psi))$ was computed completely, stop and return "**false**".*

This semi-decision algorithm can also be aborted with "**false**", because if $Res^*(\mathcal{K}(\psi))$ is finite and does not contain the empty clause, then satisfiability of the set $\mathcal{K}(\psi)$ follows from the completeness of the resolution calculus. In general, however, $Res^*(\mathcal{K}(\psi))$ is infinite and then the above algorithm does not terminate for satisfiable sets $\mathcal{K}(\psi)$.

However, the efficiency of this algorithm can be improved even further, because up to now *all* possible resolution steps between all clauses have to be performed. So the clauses that have been created by resolution steps must also be considered. In the following section we show that the resolution calculus can be further restricted to *linear* resolution without losing completeness.

3.5 Restrictions of Resolution

The problem with proving unsatisfiability by resolution is that there are many possible resolution steps, which leads to a combinatorial explosion. Therefore, our goal is to reduce the search space by using special resolution strategies without losing completeness. In this section we will present three restrictions of the resolution calculus:

- In Section 3.5.1 we present *linear* resolution, which is complete for arbitrary sets of clauses.

- In Section 3.5.2, we restrict linear resolution further to *input resolution*. This restricted form of resolution is no longer complete for arbitrary sets of clauses, but only for so-called *Horn clauses*. For this reason, the clauses used in logic programs are only *Horn clauses*.

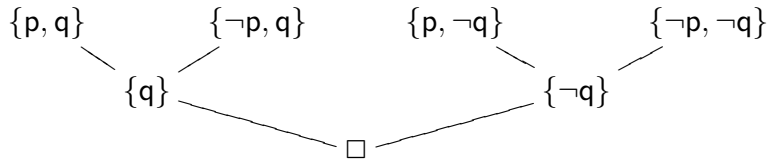
- On sets of Horn clauses, input resolution can be restricted further to *SLD resolution*. This is the resolution principle used in logic programming.

3.5.1 Linear Resolution

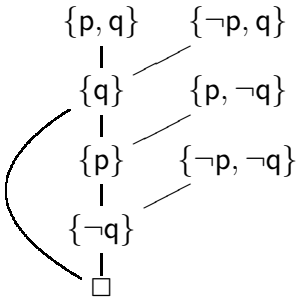
The idea behind linear resolution is that in every resolution step one of the two parent clauses must be the resolvent that was derived in the previous resolution step.

Definition 3.5.1 (Linear Resolution) *Let $\mathcal{K}$ be a set of clauses. The empty clause $\square$ can be derived from the clause K in $\mathcal{K}$ by* linear resolution *iff there is a sequence of clauses $K_1, \ldots, K_m$ with $K_1 = K \in \mathcal{K}$ and $K_m = \square$ such that for all $2 \leq i \leq m$, K_i is a resolvent of K_{i-1} and a clause from $\{K_1, \ldots, K_{i-1}\} \cup \mathcal{K}$.*

Example 3.5.2 *We consider the following set of (propositional) clauses with $\mathsf{p}, \mathsf{q} \in \Delta_0$, whose unsatisfiability can be shown with the following resolution proof:*



But this derivation does not correspond to a linear *resolution proof, because if one obtains the resolvent $\{\mathsf{q}\}$ in the first step, then this resolvent would have to be one of the parent clauses in the next step. A derivation of the empty clause with linear resolution would be possible in this example as follows:*



The following theorem shows that linear resolution is also sound and complete. So in the resolution algorithm, the computation of $Res^*(\mathcal{K}(\psi))$ can now be restricted by only considering linear resolution sequences that start with a clause from $\mathcal{K}(\psi)$.

Theorem 3.5.3 (Soundness and Completeness of Linear Resolution) *Let $\mathcal{K}$ be a set of clauses. Then $\mathcal{K}$ is unsatisfiable iff $\square$ can be derived by linear resolution from some clause K in $\mathcal{K}$. If $\mathcal{K}$ is a* minimal *unsatisfiable clause set (i.e., if for every $K \in \mathcal{K}$ the set $\mathcal{K} \setminus \{K\}$ is satisfiable), then $\square$ can be derived by linear resolution from* every *clause K in $\mathcal{K}$.*

Proof. Soundness (i.e., the direction "$\Leftarrow$") follows immediately from Theorem 3.4.10, because every linear resolution step is a resolution step.

We now show *completeness* (i.e., the direction "$\Rightarrow$"). Here, we first prove the completeness of linear *ground resolution*, i.e., we consider the case that $\mathcal{K}$ is a set of variable-free clauses.

Completeness of linear resolution in propositional logic

Let $\mathcal{K}_{min} \subseteq \mathcal{K}$ be a *minimal* unsatisfiable subset of $\mathcal{K}$, i.e., for every $K \in \mathcal{K}_{min}$, $\mathcal{K}_{min} \setminus \{K\}$ is satisfiable. Clearly $\mathcal{K}_{min} \neq \varnothing$ holds, since the empty clause set is satisfiable (even valid). We now show that $\square$ can be derived by linear resolution from *every* clause K in $\mathcal{K}_{min}$. Here, we use induction on the number n of different atomic formulas in the set $\mathcal{K}_{min}$.

Induction base: $n = 0$

We have $\mathcal{K}_{min} = \{\square\}$ and therefore $\square$ can be derived by linear resolution from the only clause $\square$ in $\mathcal{K}_{min}$.

Induction Step, Case 1: $n > 0$, $K \in \mathcal{K}_{min}$ with $|K| = 1$

Here, $K = \{L\}$ holds for some literal L. The clause set $\mathcal{K}^+$ is obtained from $\mathcal{K}_{min}$ by removing all clauses that contain L and by deleting $\overline{L}$ from all remaining clauses. As in the proof of Theorem 3.3.7, unsatisfiability of $\mathcal{K}_{min}$ also implies unsatisfiability of the set $\mathcal{K}^+$. Moreover, $\mathcal{K}^+$ contains at most $n-1$ different atomic formulas. Let $\mathcal{K}^+_{min}$ be a minimal unsatisfiable subset of $\mathcal{K}^+$. Then the induction hypothesis implies that $\square$ can be derived by linear resolution from *every* clause in $\mathcal{K}^+_{min}$.

Since $\mathcal{K}_{min}$ is a *minimal* unsatisfiable clause set, we have $\mathcal{K}^+_{min} \not\subseteq \mathcal{K}_{min}$. Thus, there is a clause $K^+ \in \mathcal{K}^+_{min}$ with $K^+ \cup \{\overline{L}\} \in \mathcal{K}_{min}$. By the induction hypothesis, $\square$ can be derived from K^+ in $\mathcal{K}^+_{min}$ by linear resolution. Thus, there is a sequence of clauses $K_1, \ldots, K_m$, such that $K_1 = K^+$ and $K_m = \square$ and such that for all $2 \leq i \leq m$, K_i is a resolvent of K_{i-1} and a clause in $\{K_1, \ldots, K_{i-1}\} \cup \mathcal{K}^+$. From this sequence we construct a sequence $K, \ldots, K_1, \ldots, K_2, \ldots, K_m = \square$ that is obtained by linear resolution steps from $\mathcal{K}_{min}$ as follows:

We show by induction on i that for all $1 \leq i \leq m$, linear resolution in $\mathcal{K}_{min}$ leads to a sequence $K, \ldots, K_i$. For $i = 1$, K, K_1 is a sequence obtained by resolution, because the resolution of $K = \{L\}$ and $K_1 \cup \{\overline{L}\}$ results in K_1. For $i > 1$, linear resolution in $\mathcal{K}_{min}$ already yields a sequence $K, \ldots, K_{i-1}$. If K_i is a resolvent of K_{i-1} and a K_j with $j < i$, then $K, \ldots, K_{i-1}, K_i$ is a sequence obtained by linear resolution steps in $\mathcal{K}_{min}$. Otherwise, K_i is a resolvent of K_{i-1} and a clause $K' \in \mathcal{K}^+$. If $K' \in \mathcal{K}_{min}$ holds as well, then $K, \ldots, K_{i-1}, K_i$ is again a sequence obtained by linear resolution steps in $\mathcal{K}_{min}$. Otherwise, $K' \cup \{\overline{L}\} \in \mathcal{K}_{min}$ holds by definition of $\mathcal{K}^+$. Therefore $K, \ldots, K_{i-1}, K_i \cup \{\overline{L}\}, K_i$ is a sequence obtained by linear resolution steps, where the last resolution step results from the parent clauses $K_i \cup \{\overline{L}\}$ and $K = \{L\}$.

Induction Step, Case 2: $n > 0$, $K \in \mathcal{K}_{min}$ with $|K| > 1$

We choose an arbitrary literal $L \in K$ and set $K^- = K \setminus \{L\}$. The set $\mathcal{K}^-$ results from $\mathcal{K}_{min}$ by removing all clauses that contain $\overline{L}$ and by deleting L from all remaining clauses. As in the proof of Theorem 3.3.7, unsatisfiability of $\mathcal{K}_{min}$ also implies unsatisfiability of the set $\mathcal{K}^-$. Moreover, $\mathcal{K}^-$ contains at most $n-1$ different atomic formulas. Let $\mathcal{K}^-_{min}$ be a minimal unsatisfiable subset of $\mathcal{K}^-$. Then the induction hypothesis implies that $\square$ can be

derived by linear resolution from *every* clause in $\mathcal{K}^-_{min}$.

We have $K^- \in \mathcal{K}^-_{min}$. The reason is as follows: On the one hand, we have $K^- \in \mathcal{K}^-$, since K does not contain the literal $\overline{L}$. (Otherwise, $L, \overline{L} \in K$ would hold and then K would be valid and would not occur in the *minimal* unsatisfiable clause set $\mathcal{K}_{min}$.) Moreover, w.l.o.g. $K^- \in \mathcal{K}^-_{min}$ holds as well, since $\mathcal{K}^- \setminus \{K^-\}$ is satisfiable. The reason is that $\mathcal{K}_{min} \setminus \{K\}$ must be satisfiable by minimality of $\mathcal{K}_{min}$. Thus, there is an interpretation $I \models \mathcal{K}_{min} \setminus \{K\}$. Since $\mathcal{K}_{min}$ is unsatisfiable, we have $I \not\models K$ and since $L \in K$, $I \not\models L$ holds. Therefore, $I \models \mathcal{K}_{min} \setminus \{K\}$ implies $I \models \mathcal{K}^-_{min} \setminus \{K^-\}$.

By the induction hypothesis, there is a derivation of $\square$ from K^- in $\mathcal{K}^-_{min}$ by linear resolution. Thus, there is a sequence of clauses $K_1, \ldots, K_m$, such that $K_1 = K^-$ and $K_m = \square$, and such that for all $2 \leq i \leq m$, K_i is a resolvent of K_{i-1} and a clause from $\{K_1, \ldots, K_{i-1}\} \cup \mathcal{K}^-_{min}$. By inserting L back in, from that we construct a sequence of linear resolution steps from K in $\mathcal{K}_{min}$. For the sequence $K_1 \cup \{L\}, \ldots, K_m \cup \{L\}$ we have: $K_1 \cup \{L\} = K \in \mathcal{K}_{min}$, $K_m \cup \{L\} = \{L\}$ and for all $2 \leq i \leq m$, $K_i \cup \{L\}$ is a resolvent of $K_{i-1} \cup \{L\}$ and a clause in $\{K_1, \ldots, K_{i-1}\} \cup \mathcal{K}_{min}$. Thus, the clause $\{L\}$ can be derived from K by linear resolution.

But $(\mathcal{K}_{min} \setminus \{K\}) \cup \{\{L\}\}$ is unsatisfiable. The reason is that $(\mathcal{K}_{min} \setminus \{K\}) \cup \{K\}$ is already unsatisfiable and that $L \in K$ holds. Every minimal unsatisfiable subset of $(\mathcal{K}_{min} \setminus \{K\}) \cup \{\{L\}\}$ contains the clause $\{L\}$, since $\mathcal{K}_{min} \setminus \{K\}$ is satisfiable (by minimality of $\mathcal{K}_{min}$). So because of Case 1, $\square$ can be derived by linear resolution from the clause $\{L\}$ in $\mathcal{K}_{min} \setminus \{K\}$. We append this derivation to the already existing derivation of $\{L\}$ and in this way, we obtain a linear derivation of $\square$ from the clause K in $\mathcal{K}_{min}$.

Completeness of linear resolution in predicate logic

Now we also prove completeness of linear resolution in predicate logic. We proceed in the same way as in the completeness proof for resolution in predicate logic (Theorem 3.4.10), by lifting the completeness proof for linear resolution in propositional logic to predicate logic.

Since $\mathcal{K}$ is unsatisfiable, Theorem 3.3.9 (a) implies that there is a finite unsatisfiable set of ground instances of clauses in $\mathcal{K}$. The previously shown completeness of linear resolution in propositional logic therefore implies that there is a sequence of clauses $K'_1, \ldots, K'_m$, such that K'_1 is ground instance of a clause $K \in \mathcal{K}$, $K'_m = \square$, and such that for all $2 \leq i \leq m$, K'_i is a resolvent of K'_{i-1} and a clause in $\{K'_1, \ldots, K'_{i-1}\} \cup \{\sigma(K) \mid K \in \mathcal{K}(\psi),\ \sigma \text{ is ground substitution}\}$. As in the proof of Theorem 3.4.10, by using the Lifting Lemma 3.4.8 we can generate a derivation of the empty clause from a clause K in $\mathcal{K}$ by linear resolution.

If $\mathcal{K}$ is a *minimal* unsatisfiable set of clauses, then the finite unsatisfiable set of its ground instances must also contain ground instances of every clause in $\mathcal{K}$. Now the completeness proof for linear propositional resolution implies that from every clause K in $\mathcal{K}$, one can derive the empty clause from a ground instance of K by linear resolution. By the Lifting Lemma we can again generate a derivation of the empty clause from K by linear resolution.

$\square$

3.5.2 Input and SLD Resolution

In order to reduce the number of possible resolution steps, we now restrict linear resolution even further. In linear resolution, one of the parent clauses in each resolution step must be the last resolvent. However, the other parent clause could still be chosen arbitrarily (i.e., it could be a clause from the original set of clauses or a previously obtained resolvent). We now prohibit the last option: in each step, now one must resolve between the previously obtained resolvent and one of the original "input" clauses. For this reason, this restriction is called *input resolution*.

Definition 3.5.4 (Input Resolution) *Let $\mathcal{K}$ be a set of clauses. The empty clause $\square$ can be derived from the clause K in $\mathcal{K}$ by* input resolution *iff there is a sequence of clauses $K_1, \ldots, K_m$ with $K_1 = K \in \mathcal{K}$ and $K_m = \square$ such that for all $2 \leq i \leq m$, K_i is a resolvent of K_{i-1} and a clause from $\mathcal{K}$.*

The definition immediately implies that input resolution is a special case of linear resolution, i.e., every proof by input resolution is also a linear resolution proof. Of course the other direction does not hold. Indeed, in contrast to linear resolution, input resolution is not complete, as shown in the following example.

Example 3.5.5 *Consider again the unsatisfiable set of clauses from Example 3.5.2. While we could prove its unsatisfiability with linear resolution, this is not possible with input resolution. By resolving two clauses from the original clause set, only the following clauses can be derived:*

$$\{\mathsf{q}\}, \{\neg\mathsf{q}\}, \{\mathsf{p}\}, \{\neg\mathsf{p}\}, \{\mathsf{q}, \neg\mathsf{q}\}, \{\mathsf{p}, \neg\mathsf{p}\}$$

If one of the first four derived clauses is again resolved with a clause of the original set, one of the first four clauses will be derived again. If one of the last two (valid) clauses is resolved with a clause of the original set, the clause of the original set is obtained. A derivation of the empty clause is only possible if two of the clauses created by resolution (like $\{\mathsf{q}\}$ and $\{\{\neg\mathsf{q}\}$) are resolved with each other. However, this is not allowed for input resolution.

So while input resolution is not complete for sets of arbitrary clauses, it is nevertheless complete on the restricted set of so-called *Horn clauses*. For this reason, logic programming does not use arbitrary clauses, but only Horn clauses.

Definition 3.5.6 (Horn Clause) *A clause is a* Horn clause *iff it contains at most one positive literal (i.e., at most one of its literals is an atomic formula and the other literals are negated atomic formulas). A Horn clause is* negative *iff it only contains negative literals (i.e., it has the form $\{\neg A_1, \ldots, \neg A_k\}$ for atomic formulas $A_1, \ldots, A_k$). A Horn clause is* definite *iff it contains a positive literal (i.e., it has the form $\{B, \neg C_1, \ldots, \neg C_n\}$ for atomic formulas $B, C_1, \ldots, C_n$).*

Thus, a set of definite Horn clauses thus corresponds to a conjunction of implications. For example, the Horn clause

$$\{\ \{\mathsf{p}, \neg\mathsf{q}\},\ \{\neg\mathsf{r}, \neg\mathsf{p}, \mathsf{s}\},\ \{\mathsf{s}\}\ \}$$

is equivalent to the formula

$$((\mathsf{p} \vee \neg \mathsf{q}) \ \wedge \ (\neg \mathsf{r} \vee \neg \mathsf{p} \vee \mathsf{s}) \ \wedge \ \mathsf{s})$$

and therefore to the following formula:

$$((\mathsf{q} \rightarrow \mathsf{p}) \ \wedge \ (\mathsf{r} \wedge \mathsf{p} \rightarrow \mathsf{s}) \ \wedge \ \mathsf{s})$$

One can see the connection to logic programming:

- *Facts* are definite Horn clauses without negative literals (i.e., they contain exactly one positive literal). An example is the clause $\{\mathsf{s}\}$. In logic programming, we write:

  ```
  s.
  ```

- *Rules* are definite Horn clauses with negative literals. An example is the clause $\{\neg \mathsf{r}, \neg \mathsf{p}, \mathsf{s}\}$. In logic programming, we write:

  ```
  s :- r,p.
  ```

- *Queries* correspond to negative Horn clauses. An example is the clause $\{\neg \mathsf{p}, \neg \mathsf{q}\}$. In logic programming, we write:

  ```
  ?- p,q.
  ```

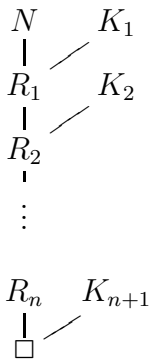
Considering only Horn clauses is a real restriction, because there is not an equivalent set of Horn clauses for every set of clauses. This already holds in propositional logic, as can be seen by the clause $\{\mathsf{p}, \mathsf{q}\}$. But in practice, the restriction to Horn clauses is often sufficient. The advantage of this restriction is that the automatic satisfiability check can be done much more efficiently. In propositional logic, satisfiability of sets of Horn clauses is decidable in polynomial time (see, e.g., [Gra11, Satz 1.12]), while it is well known that the satisfiability problem for arbitrary propositional formulas is NP complete.

For Horn clauses in predicate logic, satisfiability is still undecidable. But also in predicate logic, the restriction to Horn clauses increases efficiency considerably. The reason is that we can now restrict ourselves to input resolution, since input resolution is complete for sets of Horn clauses.

For Horn clauses, we can restrict input resolution even further to *SLD resolution* without losing completeness.

Definition 3.5.7 (SLD Resolution) *Let $\mathcal{K}$ be a set of Horn clauses with $\mathcal{K} = \mathcal{K}^d \uplus \mathcal{K}^n$, where $\mathcal{K}^d$ contains the definite clauses and $\mathcal{K}^n$ contains the negative clauses from $\mathcal{K}$. The empty clause $\square$ can be derived from the clause K in $\mathcal{K}^n$ by* SLD Resolution *iff there is a sequence of clauses $K_1, \ldots, K_m$ with $K_1 = K \in \mathcal{K}^n$ and $K_m = \square$ such that for all $2 \leq i \leq m$, K_i is a resolvent of K_{i-1} and a clause from $\mathcal{K}^d$.*

It is obvious that for SLD resolution, all clauses $K_1, \ldots, K_m$ are negative. Thus, SLD resolution proofs have the following form:



Here, $K_1, \ldots, K_{n+1} \in \mathcal{K}^d$ are definite Horn clauses from the input set, $N \in \mathcal{K}^n$ is a negative Horn clause from the input set, and the resolvents $R_1, \ldots, R_n$ are also negative Horn clauses.

The abbreviation "SLD" stands for "**l**inear resolution with **s**election function for **d**efinite clauses". In the resolution step from N to R_1 resp. in the resolution step from R_{i-1} to R_i, the "selection function" is supposed to choose the literals from N resp. R_{i-1} that are used for the resolution. At the moment, we ignore this selection function and allow resolution with any literals in the definition above. (So at the moment, we actually consider the so-called "LUSH Resolution", where "LUSH" stands for "**l**inear resolution with **u**nrestricted **s**election for **H**orn clauses"). Besides the choice of the literal that is used for the resolution, the other option in SLD resolution is the choice of the input clause from $\mathcal{K}^d$ that is used. In Section 4.3. we will discuss in more detail how these two options are restricted further in logic programming.

The following theorem shows the completeness of SLD resolution on Horn clauses.

Theorem 3.5.8 (Soundness and Completeness of SLD Resolution) *Let $\mathcal{K}$ be a set of Horn clauses. Then $\mathcal{K}$ is unsatisfiable iff $\square$ can be derived from some negative clause in $\mathcal{K}$ by SLD resolution.*

Proof. The soundness (i.e., the direction "$\Leftarrow$") is again obvious with Theorem 3.4.10, since every SLD resolution step is also a resolution step.

We now show the *completeness* (i.e., the direction "$\Rightarrow$"). Let $\mathcal{K}_{min}$ be a minimal unsatisfiable subset of $\mathcal{K}$. Any set of definite Horn clauses is satisfiable, because the interpretation which satisfies *all* atomic formulas is also a model of any definite Horn clause. Therefore $\mathcal{K}_{min}$ must also contain a negative Horn clause N. According to the completeness theorem for linear resolution (Theorem 3.5.3), $\square$ can be derived from each clause of $\mathcal{K}_{min}$ by linear resolution. So there is also a linear resolution proof which derives $\square$ from the negative clause N.

This linear resolution proof is also an SLD resolution proof. The reason is that all resolvents are negative clauses and that negative clauses cannot be resolved with negative clauses (but only with definite clauses). $\square$

Now the semi-decision algorithm from Section 3.4 for checking entailment resp. unsatisfiability can be improved. For a set of clauses that consists only of Horn clauses, we only

compute all SLD resolution sequences, but we have to start with each negative clause. If the set of clauses also contains non-Horn clauses, we have to compute all linear resolution sequences.

SLD resolution for sets of Horn clauses forms the operational basis for logic programming, as will be shown in the next chapter. However, in logic programming, one uses the restriction that each resolution step only resolves between *two* instead of arbitrary many literals. In the previous definition of resolution in predicate logic, the literals $L_1, \ldots, L_m$ from the first parent clause and the literals $L'_1, \ldots, L'_n$ from the second parent clause are deleted, if $\{\{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$ is unifiable. Instead, we now use a restriction where $m = n = 1$, called *binary* resolution. We show in Theorem 3.5.10 that this restriction preserves the completeness of resolution on Horn clauses. In contrast, binary resolution is not complete for arbitrary clauses.

Example 3.5.9 *A counterexample to the completeness of binary resolution is the following set of clauses. Here, we have* $\mathsf{p} \in \Delta_1$.

$$\{\ \{\mathsf{p}(X), \mathsf{p}(Y)\},\ \{\neg\mathsf{p}(U), \neg\mathsf{p}(V)\}\ \}$$

By using the mgu $\{X/V, Y/V, U/V\}$ *of the set* $\{\{\mathsf{p}(X), \mathsf{p}(Y), \mathsf{p}(U), \mathsf{p}(V)\}$*, the empty clause can be derived in one step with normal resolution in predicate logic. Thus, the set of clauses is unsatisfiable. In contrast, with binary resolution, only clauses like* $\{\mathsf{p}(Y), \neg\mathsf{p}(V)\}$ *can be derived. If one resolves two of these clauses, one obtains another such clause. If one of these clauses is resolved with an input clause, one gets a clause equivalent to the input clause. So the empty clause is not derivable. However, the clause* $\{\mathsf{p}(X), \mathsf{p}(Y)\}$ *is not a Horn clause.*

Theorem 3.5.10 (Soundness and Completeness of Binary SLD Resolution) *Let* $\mathcal{K}$ *be a set of Horn clauses. Then* $\mathcal{K}$ *is unsatisfiable iff* $\square$ *can be derived from a negative clause in* $\mathcal{K}$ *by binary SLD resolution.*

Proof. Soundness (i.e., the direction "$\Leftarrow$") is again obvious with Theorem 3.4.10, since every binary SLD resolution step is also a resolution step.

We now show *completeness* (i.e., the direction "$\Rightarrow$") and proceed in two steps. First we show that every resolution step with general SLD resolution can be replaced by a sequence of *unrestricted* binary SLD resolution steps. Here, *unrestricted* SLD resolution means that arbitrary unifiers can be used instead of most general unifiers. In the second step, we prove that for every unsatisfiability proof with unrestricted binary SLD resolution there is also an unsatisfiability proof with binary SLD resolution. Since general SLD resolution is complete for Horn clauses (Theorem 3.5.8), this implies the claim of the theorem.

We start with showing that every resolution step with general SLD resolution can be replaced by a sequence of *unrestricted* binary SLD resolution steps. A general SLD resolution step resolves a negative Horn clause $N = \{\neg A_1, \ldots, \neg A_m, \neg A_{m+1}, \ldots, \neg A_p\}$ with a definite clause $K = \{B, \neg C_1, \ldots, \neg C_n\}$ using the (w.l.o.g. idempotent) mgu σ of the set $\{A_1, \ldots, A_m, \nu_1(B)\}$. This results in the resolvent $R = \sigma(\{\neg A_{m+1}, \ldots, \neg A_p, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_n)\})$. Here, ν_1 is a suitable variable renaming. Because σ is mgu of $\{A_1, \ldots, A_m, B\}$,

σ is also a unifier of A_1 and B (although not necessarily their mgu). Thus, with unrestricted binary SLD resolution, we can derive the resolvent

$$R_1 = \sigma(\{\neg A_2, \ldots, \neg A_m, \neg A_{m+1}, \ldots, \neg A_p, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_n)\})$$

from N and K. Now we use another variable renaming ν_2 and extend σ such that $\sigma \circ \nu_2 = \sigma \circ \nu_1$. This is possible because w.l.o.g., ν_2 only introduces new variables that were not in the domain of σ. Then σ is a unifier of $\sigma(A_2)$ and $\nu_2(B)$. By using unrestricted binary SLD resolution again, we can therefore resolve R_1 and K and obtain the resolvent

$$R_2 = \sigma(\{\neg A_3, \ldots, \neg A_m, \neg A_{m+1}, \ldots, \neg A_p, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_n)\})$$

(since $\sigma(\sigma(A_i)) = \sigma(A_i)$ and $\sigma(\sigma(\nu_1(C_i))) = \sigma(\nu_2(C_i))$). Hence, after m unrestricted binary SLD resolution steps we obtain the clause

$$R = \sigma(\{\neg A_{m+1}, \ldots, \neg A_p, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_n)\}).$$

Now we show that every derivation of the empty clause by unrestricted binary SLD resolution in n steps can be transformed into a proof with binary SLD resolution. To this end, we use induction on n. This is trivial if $n = 0$. Otherwise, in the first step a negative Horn clause $N = \{\neg A_1, \ldots, \neg A_m\}$ is resolved with a definite clause $K_1 = \{B, \neg C_1, \ldots, \neg C_p\}$ which yields the resolvent

$$R_1 = \sigma_1(\{\neg A_2, \ldots, \neg A_m, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_p)\})$$

by using the (not necessarily most general) unifier σ_1 of the set $\{A_1, \nu_1(B)\}$. If $R_1 = \square$, then obviously we could have used the corresponding mgu instead of the unifier σ_1. Otherwise, in the next step we resolve with a definite clause $K_2 = \{D, \neg E_1, \ldots, \neg E_q\}$. Here, let σ_2 be a unifier of $\sigma_1(A_2)$[3] and $\nu_2(D)$. Then this yields the resolvent

$$R_2 = \sigma_2(\sigma_1(\{\neg A_3, \ldots, \neg A_m, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_p)\}) \cup \{\neg\nu_2(E_1), \ldots, \neg\nu_2(E_q)\}).$$

Let θ be mgu of $\{A_1, \nu_1(B)\}$. Then there is a substitution δ with $\sigma_1 = \delta \circ \theta$. With binary resolution we would have obtained the clause

$$R_1' = \theta(\{\neg A_2, \ldots, \neg A_m, \neg\nu_1(C_1), \ldots, \neg\nu_1(C_p)\})$$

from N and K_1. With another unrestricted binary SLD resolution step we could resolve R_1' and K_2. To unify $\theta(A_2)$ and $\nu_2(D)$, we can use the unifier $\sigma_2 \circ \delta$, because $\sigma_2(\delta(\theta(A_2))) = \sigma_2(\sigma_1(A_2)) = \sigma_2(\nu_2(D)) = \sigma_2(\delta(\nu_2(D)))$ (since w.l.o.g., δ's domain does not contain variables that are introduced by ν_2). Thus, by unrestricted binary resolution of R_1' and K_2 one again obtains the resolvent R_2. Thus, we can derive the empty clause in $n-1$ steps by unrestricted binary SLD resolution from R_1'. By the induction hypothesis, this is then also possible by binary SLD resolution. Since the first step from N to R_1' is a binary resolution step, the whole proof can be done by binary SLD resolution. $\square$

[3]Instead, we could also resolve with $\nu_1(C_1)$. The proof in this case would be analogous.

Chapter 4

Logic Programs

In this chapter we introduce (pure) logic programs and define their syntax and semantics in Section 4.1. Of course, this is based on the basics of predicate logic and on the resolution calculus from the previous chapters. Afterwards we show in Section 4.2 that logic programming is *universal* (or "Turing complete") i.e., that all computable programs can be implemented in this programming language. Finally, in Section 4.3 we discuss the indeterminisms in the evaluation of logic programs.

4.1 Syntax and Semantics of Logic Programs

The following definition introduces logic programs formally. Again, we use Horn clauses and sets of Horn clauses. In contrast to the previous chapter, now the order of the literals in a clause and the order of the clauses in a set is important. Therefore, from now on we consider *sequences* instead of sets. So when we speak of "clauses", then we mean *sequences* of literals and when we speak of "sets of clauses", we mean *sequences* of clauses. We keep the previous notation, i.e., we continue to write clauses and sets of clauses with curly brackets $\{\ldots\}$. However, the order of the literals or clauses is no longer irrelevant. Moreover, a clause can contain the same literal several times and a set of clauses can contain the same clause more than once.

Definition 4.1.1 (Syntax of Logic Programs) *A non-empty finite set $\mathcal{P}$ of definite Horn clauses over a signature (Σ, Δ) is called a* logic program *over (Σ, Δ). The clauses from $\mathcal{P}$ are also called* program clauses *. We distinguish the following types of program clauses:*

- Facts *(or "fact clauses") are clauses of the form $\{B\}$, where B is an atomic formula*
- Rules *(or "procedure clauses") are clauses of the form $\{B, \neg C_1, \ldots, \neg C_n\}$ with $n \geq 1$*

Here, B and $C_1, \ldots, C_n$ are atomic formulas. To execute the logic program, we pose a

- query *(or "target clause") G of the form $\{\neg A_1, \ldots, \neg A_k\}$ with $k \geq 1$*

As before, a clause represents the universally quantified disjunction of its literals and a set of clauses (e.g., a logic program) represents the conjunction of its clauses. When executing a logic program $\mathcal{P}$ with the query $G = \{\neg A_1, \ldots, \neg A_k\}$, one tries to show the following:

$$\mathcal{P} \models \exists X_1, \ldots, X_p \; A_1 \wedge \ldots \wedge A_k \tag{4.1}$$

Here, $X_1, \ldots, X_p$ are the variables in G. As mentioned above, the variables in the program clauses are implicitly universally quantified and the variables in the queries are implicitly existentially quantified. As shown in Lemma 3.0.1, the above entailment relation is equivalent to unsatisfiability of the set $\mathcal{P} \cup \{G\}$, i.e., to the unsatisfiability of $\mathcal{P} \cup \{\forall X_1, \ldots, X_p \; \neg A_1 \vee \ldots \vee \neg A_k\}$.

According to Theorem 3.3.9 (a), unsatisfiability of $\mathcal{P} \cup \{G\}$ is equivalent to the existence of a finite unsatisfiable subset of ground instances of the clauses from $\mathcal{P} \cup \{G\}$. This set cannot only consist of definite clauses, i.e., it also contains at least one ground instance of G. On the other hand, we know from the completeness of SLD resolution for Horn clauses (Theorem 3.5.8) that only one negative clause is needed to derive the empty clause from an unsatisfiable set of Horn clauses. Thus, unsatisfiability of $\mathcal{P} \cup \{G\}$ means that there is a finite unsatisfiable set of ground instances of the clauses of $\mathcal{P} \cup \{G\}$, which contains exactly one ground instance of G. Hence, (4.1) is equivalent to the existence of a set of ground terms $t_1, \ldots, t_p$ such that

$$\begin{array}{l} \mathcal{P} \cup \{(\neg A_1 \vee \ldots \vee \neg A_k)[X_1/t_1, \ldots, X_p/t_p]\} \text{ is unsatisfiable or, equivalently,} \\ \mathcal{P} \models A_1 \wedge \ldots \wedge A_k \, [X_1/t_1, \ldots, X_p/t_p]. \end{array}$$

When executing logic programs, the aim is not only to check the entailment relation from (4.1), but one wants to compute the ground terms $t_1, \ldots, t_p$, which represent the valid "solutions" for the query. Substitutions where the desired query follows from the program clauses are called *answer substitutions*. Here, the substitution is restricted to the variables of the query. If the answer substitution instantiates the variables of the query by terms with variables, the remaining variables can be replaced by any terms. The answer substitutions are generated during the SLD resolution proof, whenever the empty clause $\square$ is derived.

Example 4.1.2 *We consider a subset of the logic program from Chapter 1, i.e., a subset of the set of formulas from Example 2.1.7:*

```
motherOf(renate,susanne).
married(gerd,renate).
fatherOf(V,K) :- married(V,F), motherOf(F,K).
```

If we write this logic program $\mathcal{P}$ as a set of clauses, this results in

$$\begin{array}{ll} \{ & \{\mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne})\}, \\ & \{\mathsf{married}(\mathsf{gerd}, \mathsf{renate})\}, \\ & \{\mathsf{fatherOf}(V, F), \neg\mathsf{married}(V, F), \neg\mathsf{motherOf}(F, K)\} \quad \}. \end{array}$$

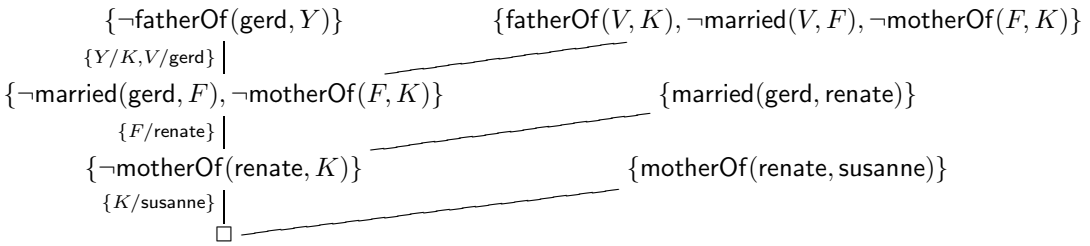
Consider the query

```
?- fatherOf(gerd,Y).
```

This means that the following negative Horn clause G is added to the above set of definite Horn clauses.

$$\{\neg\mathsf{fatherOf}(\mathsf{gerd}, Y)\}$$

We now get the following SLD resolution proof to derive the empty clause. In each resolution step, we stated the mgu which was applied to the negative parent clause and the (possibly variable-renamed) program clause.



The answer substitution is now obtained by composing the individual substitutions

$$\{K/\mathsf{susanne}\} \circ \{F/\mathsf{renate}\} \circ \{Y/K, V/\mathsf{gerd}\} = \{K/\mathsf{susanne}, F/\mathsf{renate}, Y/\mathsf{susanne}, V/\mathsf{gerd}\}$$

and then restricting it to the variables of the query. Since the query in our example only contains the variable Y, the answer substitution is $\{Y/\mathsf{susanne}\}$.

We now define the semantics of logic programs. More precisely, we will define three different ways to define the semantics and we will prove that all three possibilities are equivalent. These different forms are called *declarative*, *procedural*, and *fixpoint* semantics.

4.1.1 Declarative Semantics of Logic Programming

The idea of the declarative (or "model theoretical") semantics is to consider the logic program as a static "data base" and to define the "true" statements about the program using the entailment relation of predicate logic. More precisely, we define the semantics of a program $\mathcal{P}$ *w.r.t.* a query G. The declarative semantics consists of all ground instances of G which are "true statements" about the program $\mathcal{P}$.

Definition 4.1.3 (Declarative Semantics of a Logic Program) *Let $\mathcal{P}$ be a logic program and $G = \{\neg A_1, \ldots, \neg A_k\}$ a query. Here A_1, ..., A_k are atomic formulas. Then the* declarative semantics *of $\mathcal{P}$ with respect to G is defined as*

$$D[\![\mathcal{P}, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid \mathcal{P} \models \sigma(A_1 \wedge \ldots \wedge A_k),\ \sigma \textit{ is ground substitution}\}.$$

Each ground instance $\sigma(A_1 \wedge \ldots \wedge A_k)$ in $D[\![\mathcal{P}, G]\!]$ contains the ground substitution of the variables from A_1, ..., A_k as the corresponding "solution".

Example 4.1.4 *Again we regard the logic program $\mathcal{P}$ and the query $G = \{\neg\mathsf{fatherOf}(\mathsf{gerd}, Y)\}$ from Example 4.1.2. The only ground instance of $\mathsf{fatherOf}(\mathsf{gerd}, Y)$ that is entailed by $\mathcal{P}$ is $\mathsf{fatherOf}(\mathsf{gerd}, \mathsf{susanne})$ (i.e., $\mathcal{P} \models \mathsf{fatherOf}(\mathsf{gerd}, \mathsf{susanne})$). Thus*

$$D[\![\mathcal{P}, G]\!] = \{\mathsf{fatherOf}(\mathsf{gerd}, \mathsf{susanne})\}.$$

If $\mathcal{P}$ contained the additional fact $\mathsf{motherOf}(\mathsf{renate}, \mathsf{peter})$, then we would have

$$D[\![\mathcal{P}, G]\!] = \{\mathsf{fatherOf}(\mathsf{gerd}, \mathsf{susanne}), \mathsf{fatherOf}(\mathsf{gerd}, \mathsf{peter})\}.$$

4.1.2 Procedural Semantics of Logic Programming

The procedural (or "operational") semantics "operationalizes" the declarative semantics by explicitly specifying how the corresponding entailment from $\mathcal{P}$ is checked. To this end, one uses SLD resolution and keeps track of the substitutions that are applied in each resolution step. More precisely, we specify an abstract *interpreter* for logic programs which operates on so-called *configurations*. A configuration is a pair of a query (i.e., a negative clause) and a substitution. We start with the configuration $(G, \varnothing)$ consisting of the original query and the identical substitution $\varnothing$. The goal is to reach a final configuration of the form $(\square, \sigma)$. In this case, σ (or the restriction of σ to the variables of the original query G) is the computed *answer substitution*. A *computation* is a sequence of configurations, where the definition of the interpreter determines which transitions from one configuration to the next are allowed.

The form of the SLD resolution used in logic programs differs from the general SLD resolution (Definition 3.5.7) in three aspects.

- Instead of renaming variables in both parent clauses we restrict ourselves to the so-called *standardized* SLD resolution. Here, a variable renaming may only be applied to the (definite) program clauses, but not to the other (negative) parent clause. Of course, such a restriction is possible without loss of generality.

- In the resolution step, we only resolve two literals instead of an arbitrary number of literals. So we only use *binary* SLD Resolution. However, as shown in Theorem 3.5.10, this restriction preserves completeness for Horn clauses.

- Finally, we do not consider clauses as sets anymore, but as *sequences* of literals. This means that in such a sequence, a literal can occur several times. For example, the resolution of the clauses $\{\neg\mathsf{p}, \neg\mathsf{p}\}$ and $\{\mathsf{p}\}$ results in the resolvent $\{\neg\mathsf{p}\}$, where $\mathsf{p} \in \Delta_0$. Similar to the completeness proof for binary SLD resolution (Theorem 3.5.10), binary SLD resolution for these sequences is also complete. The reason is that obviously every resolution step on sets can be replaced by a sequence of resolution steps on sequences.

Definition 4.1.5 (Procedural Semantics of a Logic Program) *Let $\mathcal{P}$ be a logic program.*

- *A* configuration *is a pair (G, σ), where G is a query or the empty clause $\square$, and where σ is a substitution.*

- *There is a* computation step $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ *iff*
  - $G_1 = \{\neg A_1, \ldots, \neg A_k\}$ *with* $k \geq 1$
  - *there are a program clause* $K \in \mathcal{P}$ *and a variable renaming* ν *with* $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ *and* $n \geq 0$*, such that*
    - $\nu(K)$ *does not have any variables in common with* G_1 *or* $RANGE(\sigma_1)$ *and*
    - *there is an* $1 \leq i \leq k$ *such that* A_i *and* B *are unifiable with an mgu* σ
  - $G_2 = \sigma(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\})$
  - $\sigma_2 = \sigma \circ \sigma_1$
- *A* computation *of* $\mathcal{P}$ *for the query* $G = \{\neg A_1, \ldots, \neg A_k\}$ *is a (finite or infinite) sequence of configurations of the form*

$$(G, \varnothing) \vdash_{\mathcal{P}} (G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \ldots$$

- *A computation terminating with* $(\square, \sigma)$*, which started with* $(G, \varnothing)$ *(where* $G = \{\neg A_1, \ldots, \neg A_k\}$*), is called* successful *with the computation result* $\sigma(A_1 \wedge \ldots \wedge A_k)$*. The computed* answer substitution *is* σ *restricted to the variables of* G*.*

The procedural semantics *of* $\mathcal{P}$ *w.r.t.* G *is defined as*

$$P[\![\mathcal{P}, G]\!] = \{\sigma'(A_1 \wedge \ldots \wedge A_k) \mid (G, \varnothing) \vdash^+_{\mathcal{P}} (\square, \sigma), \\ \sigma'(A_1 \wedge \ldots \wedge A_k) \text{ is ground instance } \sigma(A_1 \wedge \ldots \wedge A_k)\}.$$

Here "$\vdash^+_{\mathcal{P}}$*" is the transitive closure of "*$\vdash_{\mathcal{P}}$*" (i.e.,* $(G, \varnothing) \vdash^+_{\mathcal{P}} (\square, \sigma)$ *holds iff* $(G, \varnothing) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \sigma)$*). Analogously, for all* $l \in \mathbb{N}$ *we define the relation* $\vdash^l_{\mathcal{P}}$ *as* $(G, \sigma) \vdash^l_{\mathcal{P}} (G_l, \sigma_l)$ *iff there are* G_i *and* σ_i *with* $(G, \sigma) \vdash_{\mathcal{P}} (G_1, \sigma_1) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (G_l, \sigma_l)$*.*

Example 4.1.6 *The procedural semantics corresponds to the procedure used in Example 4.1.2. Here, we have:*

$$\begin{array}{ll} & (\{\neg \mathsf{fatherOf}(\mathsf{gerd}, Y)\}, \varnothing) \\ \vdash_{\mathcal{P}} & (\{\neg \mathsf{married}(\mathsf{gerd}, F), \neg \mathsf{motherOf}(F, K)\}, \{Y/K, V/\mathsf{gerd}\}) \\ \vdash_{\mathcal{P}} & (\{\neg \mathsf{motherOf}(\mathsf{renate}, K)\}, \{F/\mathsf{renate}, Y/K, V/\mathsf{gerd}\}) \\ \vdash_{\mathcal{P}} & (\square, \{K/\mathsf{susanne}, F/\mathsf{renate}, Y/\mathsf{susanne}, V/\mathsf{gerd}\}) \end{array}$$

and the answer substitution is $\{Y/\mathsf{susanne}\}$*. We obtain*

$$P[\![\mathcal{P}, G]\!] = \{\mathsf{fatherOf}(\mathsf{gerd}, \mathsf{susanne})\}.$$

The next example shows that there are still two indeterminisms in the computation steps of the procedural semantics:

- On the one hand, one has to choose the program clause K for the next resolution step.
- On the other hand, one has to select the literal A_i from the current query that should be used for the next resolution step.

Example 4.1.7 *Consider the following logic program* $\mathcal{P}$

$$\begin{array}{ll}\{ & \{\{\mathsf{p}(X,Z), \neg\mathsf{q}(X,Y), \neg\mathsf{p}(Y,Z)\},\\ & \{\mathsf{p}(U,U)\},\\ & \{\mathsf{q}(\mathsf{a},\mathsf{b})\}\ \}\end{array}$$

with $\mathsf{p},\mathsf{q} \in \Delta_2$ *and* $\mathsf{a},\mathsf{b} \in \Sigma_0$. *We analyze the query*

$$G = \{\neg\mathsf{p}(V,\mathsf{b})\}.$$

The following computation is not successful (but finite and no further computation step is possible). We always underlined the literal A_i *which was used in the respective resolution step:*

$$\begin{array}{rl} & (\{\underline{\neg\mathsf{p}(V,\mathsf{b})}\}, \varnothing)\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{q}(V,Y)}, \neg\mathsf{p}(Y,\mathsf{b})\}, \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{p}(\mathsf{b},\mathsf{b})}\}, \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{q}(\mathsf{b},Y')}, \neg\mathsf{p}(Y',\mathsf{b})\}, \{X'/\mathsf{b}, Z'/\mathsf{b}\} \circ \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\{\neg\mathsf{q}(\mathsf{b},\mathsf{b})\}, \{U/\mathsf{b}, Y'/\mathsf{b}\} \circ \{X'/\mathsf{b}, Z'/\mathsf{b}\} \circ \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\})\end{array}$$

In contrast, the following computation would be successful. Its first three steps are the same, but then we resolve with the second instead of the first program clause:

$$\begin{array}{rl} & (\{\underline{\neg\mathsf{p}(V,\mathsf{b})}\}, \varnothing)\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{q}(V,Y)}, \neg\mathsf{p}(Y,\mathsf{b})\}, \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\{\underline{\neg\mathsf{p}(\mathsf{b},\mathsf{b})}\}, \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\})\\ \vdash_{\mathcal{P}} & (\square, \underbrace{\{U/\mathsf{b}\} \circ \{V/\mathsf{a}, Y/\mathsf{b}\} \circ \{X/V, Z/\mathsf{b}\}}_{\{U/\mathsf{b},\ V/\mathsf{a},\ Y/\mathsf{b},\ X/\mathsf{a},\ Z/\mathsf{b}\}})\end{array}$$

The answer substitution is $\{V/\mathsf{a}\}$ *and thus,* $\mathsf{p}(\mathsf{a},\mathsf{b}) \in P[\![\mathcal{P}, G]\!]$.

But in this example, there is another successful computation, by using the second program clause already in the first resolution step:

$$\begin{array}{rl} & (\{\{\underline{\neg\mathsf{p}(V,\mathsf{b})}\}, \varnothing)\\ \vdash_{\mathcal{P}} & (\square, \{U/\mathsf{b}, V/\mathsf{b}\})\end{array}$$

Now the answer substitution is $\{V/\mathsf{b}\}$ *and therefore* $\mathsf{p}(\mathsf{b},\mathsf{b}) \in P[\![\mathcal{P}, G]\!]$.

The following theorem by Clark shows that the declarative and procedural semantics are equivalent. In particular, this means that the restrictions of SLD resolution do not destroy the completeness of logic programming.

Theorem 4.1.8 (Equivalence of Declarative and Procedural Semantics) *Let* $\mathcal{P}$ *be a logic program and let* $G = \{\neg A_1, \ldots, \neg A_k\}$ *be a query. Then we have* $D[\![\mathcal{P}, G]\!] = P[\![\mathcal{P}, G]\!]$.

Proof. We first show the soundness of the procedural semantics w.r.t. the declarative semantics, i.e., $P[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$. Let $\sigma'(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, G]\!]$. Then there is a successful computation of the form

$$(G, \varnothing) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \sigma),$$

where $\sigma'(A_1 \wedge \ldots \wedge A_k)$ is a ground instance of $\sigma(A_1 \wedge \ldots \wedge A_k)$. We have to show that then we have $\mathcal{P} \models \sigma'(A_1 \wedge \ldots \wedge A_k)$. We use induction on the length l of the computation. More precisely, l is the number of $\vdash_{\mathcal{P}}$-steps in the computation. So the computation has the form

$$(G, \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash_{\mathcal{P}} (G_2, \delta_2 \circ \delta_1) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1)$$

with $\sigma = \delta_l \circ \ldots \circ \delta_1$. So there is an A_i, a $K \in \mathcal{P}$, and a variable renaming ν with $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ and $n \geq 0$, such that G and $\nu(K)$ have no common variables, such that δ_1 is the mgu of A_i and B, and such that

$$G_1 = \delta_1(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}).$$

In the induction base, we have $l = 1$. Then $G_1 = \square$ and thus, $i = k = 1$, $n = 0$, $\nu(K) = \{B\}$ (i.e., the program clause K is a fact), and $\sigma = \delta_1$. Since all ground instances of B are entailed by $\mathcal{P}$ and since $\delta_1(B) = \delta_1(A_1) = \sigma(A_1)$ holds, all ground instances of $\sigma(A_1)$ are entailed by $\mathcal{P}$ as well. So in particular, $\mathcal{P} \models \sigma'(A_1)$.

In the induction step, we have $l > 1$. Obviously, then we also have the computation

$$(G_1, \varnothing) \vdash_{\mathcal{P}} (G_2, \delta_2) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_2).$$

According to the induction hypothesis, every ground instance of

$$\delta_l(\ldots \delta_2(\delta_1(A_1 \wedge \ldots \wedge A_{i-1} \wedge C_1 \wedge \ldots \wedge C_n \wedge A_{i+1} \wedge \ldots \wedge A_k)) \ldots)$$

is entailed by $\mathcal{P}$. Since all ground instances of $C_1 \wedge \ldots \wedge C_n \rightarrow B$ are also entailed by $\mathcal{P}$, every ground instance of

$$\delta_l(\ldots \delta_1(A_1 \wedge \ldots \wedge A_{i-1} \wedge B \wedge A_{i+1} \wedge \ldots \wedge A_k) \ldots)$$

is entailed by $\mathcal{P}$ as well. As $\delta_1(B) = \delta_1(A_i)$ holds, all ground instances

$$\delta_l(\ldots \delta_1(A_1 \wedge \ldots \wedge A_k) \ldots)$$

are entailed by $\mathcal{P}$, too. Since $\sigma = \delta_l \circ \ldots \circ \delta_1$, we have $\mathcal{P} \models \sigma'(A_1 \wedge \ldots \wedge A_k)$.

Now we show the completeness of the procedural semantics w.r.t. the declarative semantics, i.e., $D[\![\mathcal{P}, G]\!] \subseteq P[\![\mathcal{P}, G]\!]$. Let $\sigma(A_1 \wedge \ldots \wedge A_k) \in D[\![\mathcal{P}, G]\!]$. Then $\mathcal{P} \models \sigma(A_1 \wedge \ldots \wedge A_k)$ holds, i.e., $\mathcal{P} \cup \{\sigma(\{\{\neg A_1, \ldots, \neg A_k\})\}$ is unsatisfiable according to Lemma 3.0.1.

Due to the completeness of binary SLD resolution (Theorem 3.5.10), one can derive the empty clause $\square$ from the negative clause $\sigma(\{\neg A_1, \ldots, \neg A_k\})$ in $\mathcal{P} \cup \{\sigma(\{\neg A_1, \ldots, \neg A_k\})\}$ by binary SLD resolution. Thus, there is a successful computation

$$(\sigma(\{\neg A_1, \ldots, \neg A_k\}), \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \delta_l \circ \ldots \circ \delta_1).$$

Here, l is the length of the computation and δ_i is the mgu used in the i-th resolution step (for all $1 \leq i \leq l$).

The above computation starts with a query $\sigma(\{\neg A_1, \ldots, \neg A_k\})$ without variables, since σ is a ground substitution. We now show that this computation can be "lifted" into a computation of the form

$$(\{\neg A_1, \ldots, \neg A_k\}, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \delta'_l \circ \ldots \circ \delta'_1)$$

such that $\sigma(A_1 \wedge \ldots \wedge A_k)$ is a ground instance of $\delta'_l(\ldots\delta'_1(A_1 \wedge \ldots \wedge A_k)\ldots)$. Then, we obtain $\sigma(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, G]\!]$, as desired.

To show the above statement, we prove a more general claim for arbitrary substitutions σ, where $DOM(\sigma)$ contains only variables from G:[1]

$$\begin{array}{l}\text{If } (\sigma(G), \varnothing) \vdash^+_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1), \text{ then there is a computation } (G, \varnothing) \vdash^+_{\mathcal{P}} \\ (\square, \delta'_l \circ \ldots \circ \delta'_1) \text{ and a substitution } \sigma' \text{ with } \delta_l \circ \ldots \circ \delta_1 \circ \sigma = \sigma' \circ \delta'_l \circ \ldots \circ \delta'_1.\end{array} \tag{4.2}$$

This claim implies the above statement, since there σ is a ground substitution on all variables of G and therefore $\delta_l(\ldots\delta_1(\sigma(G))\ldots) = \sigma(G)$ holds.

We use induction on the length l of the computation. In the first step $(\sigma(G), \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1)$, there exists an $A_i \in G$ which is used for resolution with a clause $K \in \mathcal{P}$. W.l.o.g., here we use a variable renaming ν for the resolution such that $\nu(K)$ has no variables in common with G. Then we have $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ with $n \geq 0$, such that δ_1 is the mgu of $\sigma(A_i)$ and B and we have

$$G_1 = \delta_1(\{\neg\sigma(A_1), \ldots, \neg\sigma(A_{i-1}), \neg C_1, \ldots, \neg C_n, \neg\sigma(A_{i+1}), \ldots, \neg\sigma(A_k)\}).$$

We now show that a corresponding resolution step can also be performed with G and $\nu(K)$ instead of $\sigma(G)$ and $\nu(K)$. Instead of the literal $\sigma(A_i) \in \sigma(G)$ we now use the literal $A_i \in G$ for the resolution. Since σ only instantiates variables from A_i and not from B, $\delta_1 \circ \sigma$ is a unifier of A_i and B. Therefore, A_i and B also have an mgu δ'_1. So there is a substitution τ with

$$\delta_1 \circ \sigma = \tau \circ \delta'_1. \tag{4.3}$$

The resolution step with G and $\nu(K)$ results in

$$G'_1 = \delta'_1(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}).$$

So $(G, \varnothing) \vdash_{\mathcal{P}} (G'_1, \delta'_1)$. Note that the resolvent G_1 which results from $\sigma(G)$ and $\nu(K)$ is indeed an instance of the resolvent G'_1 of G and $\nu(K)$, because $\tau(G'_1) = G_1$.

In the induction base we have $l = 1$ and thus, $G_1 = G'_1 = \square$. We have $(G, \varnothing) \vdash_{\mathcal{P}} (\square, \delta'_1)$ and (4.3) implies $\delta_1 \circ \sigma = \sigma' \circ \delta'_1$ as desired, if we set $\sigma' = \tau$. Thus, (4.2) is proven.

In the induction step, we have $l > 1$. Since $(\sigma(G), \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash^+_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1)$ is a computation of length l, $(G_1, \varnothing) \vdash^+_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_2)$ is a computation of length $l - 1$. We have already shown that the first computation step $(\sigma(G), \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1)$ can be "lifted" to $(G, \varnothing) \vdash_{\mathcal{P}} (G'_1, \delta'_1)$. Since $G_1 = \tau(G'_1)$, by the induction hypothesis there is also a computation $(G'_1, \varnothing) \vdash^+_{\mathcal{P}} (\square, \delta'_l \circ \ldots \circ \delta'_2)$ and a substitution τ' with

$$\delta_l \circ \ldots \circ \delta_2 \circ \tau = \tau' \circ \delta'_l \circ \ldots \circ \delta'_2. \tag{4.4}$$

This implies the first part of the claim (4.2): $(G, \varnothing) \vdash_{\mathcal{P}} (G'_1, \delta'_1) \vdash^+_{\mathcal{P}} (\square, \delta'_l \circ \ldots \circ \delta'_2 \circ \delta'_1)$. For the second part of the claim (4.2) we have:

$$\begin{array}{lll} & \delta_l \circ \ldots \circ \delta_2 \circ \delta_1 \circ \sigma & \\ = & \delta_l \circ \ldots \circ \delta_2 \circ \tau \circ \delta'_1 & \text{by (4.3) (during 1-step lifting, } \tau \text{ results from } \sigma) \\ = & \tau' \circ \delta'_l \circ \ldots \circ \delta'_2 \circ \delta'_1 & \text{by (4.4) (during } (l-1)\text{-step lifting, } \tau' \text{ results from } \tau). \end{array}$$

[1] This claim is not a direct consequence of the Lifting Lemma 3.4.8, since here the respective parent clauses from the program contain variables and since the claim also concerns the substitutions used, which is not the case for the Lifting Lemma.

So we obtain the second part of the claim (4.2) if we set $\sigma' = \tau'$. □

4.1.3 Fixpoint Semantics of Logic Programming

In the following, we introduced a third possibility to define the semantics. More precisely, now the semantics of a logic program is defined as the fixpoint of a transformation $\underline{\text{trans}}_{\mathcal{P}}$. This transformation derives all true statements about a program step by step. In contrast to the declarative semantics, here the semantics is not defined in a model-theoretical way, but (similar to the procedural semantics) the restriction to Horn clauses is used to derive true statements by in a way that is similar to resolution. In contrast to the procedural semantics, now the derivation is not based on the query, but only on the program and it derives all true statements (without variables).

For a logic program $\mathcal{P}$, $\underline{\text{trans}}_{\mathcal{P}}$ is a function which maps sets of ground atomic formulas (i.e., without variables) to sets of ground atomic formulas. Here, $\underline{\text{trans}}_{\mathcal{P}}(M)$ contains all formulas of M and in addition, it contains all ground atomic formulas which can be derived in one step from M by applying a rule from $\mathcal{P}$. In the following, $Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$ denotes the set of all sets of atomic formulas without variables.

Definition 4.1.9 (The Transformation $\underline{\text{trans}}_{\mathcal{P}}$) *Let $\mathcal{P}$ be a logic program over a signature (Σ, Δ). The corresponding transformation $\underline{\text{trans}}_{\mathcal{P}}$ is a function*

$$\underline{\text{trans}}_{\mathcal{P}} : Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing)) \to Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$$

with

$$\begin{aligned}\underline{\text{trans}}_{\mathcal{P}}(M) = M \cup \{A' \mid\ & \{A', \neg B'_1, \ldots, \neg B'_n\} \textit{ is ground instance} \\ & \textit{of a clause } \{A, \neg B_1, \ldots, \neg B_n\} \in \mathcal{P} \\ & \textit{and } B'_1, \ldots, B'_n \in M\ \}\end{aligned}$$

The idea of defining the semantics is to start from the empty set of formulas $\varnothing$. Then $\varnothing$ describes all formulas which can be derived from $\mathcal{P}$ in zero steps and $\underline{\text{trans}}_{\mathcal{P}}(\varnothing)$ are all atomic formulas (without variables) which can be derived from $\mathcal{P}$ with at most one application of a clause. Thus, these are all ground instances of the facts of $\mathcal{P}$. Similarly, $\underline{\text{trans}}_{\mathcal{P}}(\underline{\text{trans}}_{\mathcal{P}}(\varnothing)) = \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing)$ are all ground atomic formulas that can be derived from $\mathcal{P}$ in at most two steps, etc. So the set

$$M_{\mathcal{P}} = \varnothing \cup \underline{\text{trans}}_{\mathcal{P}}(\varnothing) \cup \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing) \cup \underline{\text{trans}}^3_{\mathcal{P}}(\varnothing) \cup \ldots = \bigcup_{i \in \mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$$

contains all ground atomic formulas which can be derived from $\mathcal{P}$. Now one can define the semantics of $\mathcal{P}$ with respect to a query G by taking all ground instances of G which are contained in $\bigcup_{i \in \mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$.

Example 4.1.10 *Again, we consider the logic program $\mathcal{P}$ from Example 4.1.2.*

$$\begin{array}{ll}\{ & \{\mathsf{motherOf}(\mathsf{renate},\mathsf{susanne})\},\\ & \{\mathsf{married}(\mathsf{gerd},\mathsf{renate})\},\\ & \{\mathsf{fatherOf}(V,K),\neg\mathsf{married}(V,F),\neg\mathsf{motherOf}(F,K)\}\ \}.\end{array}$$

We obtain

$$\begin{array}{rcl}\underline{\mathrm{trans}}_{\mathcal{P}}(\varnothing) & = & \{\mathsf{motherOf}(\mathsf{renate},\mathsf{susanne}),\mathsf{married}(\mathsf{gerd},\mathsf{renate})\}\\ \underline{\mathrm{trans}}^2_{\mathcal{P}}(\varnothing) & = & \underline{\mathrm{trans}}_{\mathcal{P}}(\varnothing)\cup\{\mathsf{fatherOf}(\mathsf{gerd},\mathsf{susanne})\}\end{array}$$

and $\underline{\mathrm{trans}}^i_{\mathcal{P}}(\varnothing)=\underline{\mathrm{trans}}^2_{\mathcal{P}}(\varnothing)$ for all $i\geq 2$.So in this example, $M_{\mathcal{P}}$ is reached after only two iterations.

Example 4.1.11 *In general, the iteration for computing $M_{\mathcal{P}}$ can be infinite. To this end, consider the following logic program:*

```
p(a).
p(f(X)) :- p(X).
```

It corresponds to the following set of clauses:

$$\begin{array}{ll}\{ & \{\mathsf{p}(\mathsf{a})\},\\ & \{\mathsf{p}(\mathsf{f}(X)),\neg\mathsf{p}(X)\}\ \}\end{array}$$

Now we have

$$\begin{array}{rcl}\underline{\mathrm{trans}}_{\mathcal{P}}(\varnothing) & = & \{\mathsf{p}(\mathsf{a})\},\\ \underline{\mathrm{trans}}^2_{\mathcal{P}}(\varnothing) & = & \{\mathsf{p}(\mathsf{a}),\mathsf{p}(\mathsf{f}(\mathsf{a}))\},\\ \underline{\mathrm{trans}}^3_{\mathcal{P}}(\varnothing) & = & \{\mathsf{p}(\mathsf{a}),\mathsf{p}(\mathsf{f}(\mathsf{a})),\mathsf{p}(\mathsf{f}^2(\mathsf{a}))\},\ \textit{etc.}\end{array}$$

This results in $\underline{\mathrm{trans}}^{i+1}_{\mathcal{P}}(\varnothing)=\{\mathsf{p}(\mathsf{a}),\mathsf{p}(\mathsf{f}(\mathsf{a})),\ldots,\mathsf{p}(\mathsf{f}^i(\mathsf{a}))\}$ and $M_{\mathcal{P}}=\bigcup_{i\in\mathbb{N}}\underline{\mathrm{trans}}^i_{\mathcal{P}}(\varnothing)=\{\mathsf{p}(\mathsf{f}^i(\mathsf{a}))\mid i\in\mathbb{N}\}$.

The above construction computes a *fixpoint* of the transformation $\underline{\mathrm{trans}}_{\mathcal{P}}$, because $\underline{\mathrm{trans}}_{\mathcal{P}}(M_{\mathcal{P}})=M_{\mathcal{P}}$ holds. In fact, $M_{\mathcal{P}}$ is the *smallest* fixpoint of $\underline{\mathrm{trans}}_{\mathcal{P}}$, if one compares sets w.r.t. the subset relation $\subseteq$. This means that $M_{\mathcal{P}}$ contains only those formulas which really have to be included if one wants to obtain all ground atomic formulas which can be derived from $\mathcal{P}$.

We now show that the set $M_{\mathcal{P}}$ computed above is really always the smallest fixpoint of $\underline{\mathrm{trans}}_{\mathcal{P}}$. This also proves that $\underline{\mathrm{trans}}_{\mathcal{P}}$ always has a (unique) smallest fixpoint.

As usual, an *order* is a transitive and antisymmetric relation. The relation $\subseteq$ on $Pot(\mathcal{A}t(\Sigma,\Delta,\varnothing))$ is obviously a reflexive order, i.e., for all $M_1,M_2,M_3\subseteq\mathcal{A}t(\Sigma,\Delta,\varnothing)$ we have:

- $M_1\subseteq M_1$ (reflexivity)
- $M_1\subseteq M_2$ and $M_2\subseteq M_3$ imply $M_1\subseteq M_3$ (transitivity)
- $M_1\subseteq M_2$ and $M_2\subseteq M_1$ imply $M_1=M_2$ (antisymmetry)

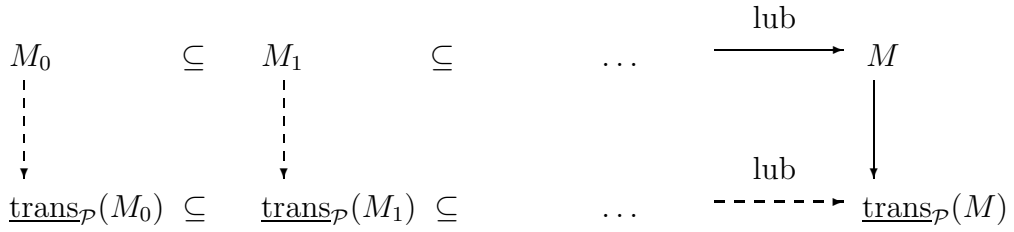


Figure 4.1: Continuity of $\underline{\text{trans}}_{\mathcal{P}}$

If one considers the sequence of sets

$$\varnothing,\ \underline{\text{trans}}_{\mathcal{P}}(\varnothing),\ \underline{\text{trans}}^2_{\mathcal{P}}(\varnothing),\ \underline{\text{trans}}^3_{\mathcal{P}}(\varnothing), \ldots$$

one notices that the sets obviously get "larger" regarding the order $\subseteq$. Such sequences are called *chains*. Then the desired set $M_{\mathcal{P}}$ is the "limit" of this chain, i.e., it is its *least upper bound* or "*lub*". A reflexive order is *complete* iff it has a smallest element and each chain has a least upper bound. Such an order is also called "*complete partial order*" or "*cpo*". The following lemma shows that $\subseteq$ on $Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$ is indeed complete. Thus, for every program $\mathcal{P}$ there exists the least upper bound $M_{\mathcal{P}}$ of the above chain.[2]

Lemma 4.1.12 (Completeness of $\subseteq$) *The relation $\subseteq$ is complete on $Pot(\mathcal{A}t(\Sigma, \Delta, \varnothing))$: The smallest element of $\mathcal{A}t(\Sigma, \Delta, \varnothing)$ w.r.t. $\subseteq$ is the empty set $\varnothing$ and every chain*

$$M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots$$

with $M_i \subseteq \mathcal{A}t(\Sigma, \Delta, \varnothing)$ has the least upper bound $M = \bigcup_{i \in \mathbb{N}} M_i$. Thus $M_i \subseteq M$ holds for all M_i and for every other upper bound M', we have $M \subseteq M'$.

Proof. That $\varnothing$ is the smallest element is obvious, since $\varnothing \subseteq N$ for all $N \subseteq \mathcal{A}t(\Sigma, \Delta, \varnothing)$. It is also obvious that $M = \bigcup_{i \in \mathbb{N}} M_i$ is an upper bound of the chain, because trivially $M_i \subseteq M$ holds. Let now M' be another upper bound, i.e., $M_i \subseteq M'$ for all M_i. Then obviously, we also have $M = \bigcup_{i \in \mathbb{N}} M_i \subseteq M'$.

The function $\underline{\text{trans}}_{\mathcal{P}}$ has two important properties which are essential to guarantee that it always has a least fixpoint. It is *monotonic* and *continuous*. In this context, monotonicity means that more formulas can be derived from larger sets of formulas. Continuity means that $\underline{\text{trans}}_{\mathcal{P}}$ maps the limit of each chain to the same result that one obtains when computing the result for each individual element of the chain and then taking the limit of these results. This is illustrated in Figure 4.1. If one first looks at the chain $M_0 \subseteq M_1 \subseteq \ldots$, takes its limit $M = \bigcup_{i \in \mathbb{N}} M_i$, and then applies $\underline{\text{trans}}_{\mathcal{P}}$ to it (solid arrows), the result must be the same as if one first applies $\underline{\text{trans}}_{\mathcal{P}}$ to the individual elements of the chain and then takes the limit (dashed arrows).

Lemma 4.1.13 (Monotonicity and Continuity)

[2]Formal general definitions for these notions as well as for the notions "monotonicity" and "continuity" can be found, e.g., in [Gie14].

(a) The function $\underline{\text{trans}}_{\mathcal{P}}$ *is* monotonic*, i.e., if* $M_1 \subseteq M_2$*, then we also have* $\underline{\text{trans}}_{\mathcal{P}}(M_1) \subseteq \underline{\text{trans}}_{\mathcal{P}}(M_2)$.

(b) The function $\underline{\text{trans}}_{\mathcal{P}}$ *is* continuous*, i.e., for every chain*

$$M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots,$$

we have $\underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i) = \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}_{\mathcal{P}}(M_i)$.

Proof. Part (a) is obvious. We now show Part (b). Here, the monotonicity of $\underline{\text{trans}}_{\mathcal{P}}$ implies $\underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i) \supseteq \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}_{\mathcal{P}}(M_i)$. The reason is that because of the monotonicity, $\underline{\text{trans}}_{\mathcal{P}}(M_i) \subseteq \underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i)$ holds for all i.

To also show $\underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i) \subseteq \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}_{\mathcal{P}}(M_i)$, let $A' \in \underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i)$. Then $A' \in \bigcup_{i\in\mathbb{N}} M_i$ or $\{A', \neg B'_1, \ldots, \neg B'_n\}$ is a ground instance of a clause $\{A, \neg B_1, \ldots, \neg B_n\} \in \mathcal{P}$ and $B'_1, \ldots, B'_n \in \bigcup_{i\in\mathbb{N}} M_i$. Since $M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots$ is a chain, there is a $j \in \mathbb{N}$ with $A' \in M_j$ or $B'_1, \ldots, B'_n \in M_j$. Therefore, we also have $A' \in \underline{\text{trans}}_{\mathcal{P}}(M_j) \subseteq \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}_{\mathcal{P}}(M_i)$. □

Now we can show that $\underline{\text{trans}}_{\mathcal{P}}$ indeed always has a smallest fixpoint ("*least fixpoint*" or "*lfp*") and that this corresponds to the least upper bound of the chain $\varnothing$, $\underline{\text{trans}}_{\mathcal{P}}(\varnothing)$, $\underline{\text{trans}}^2_{\mathcal{P}}(\varnothing), \ldots$. The following theorem corresponds to the general Fixpoint Theorem of Tarski resp. Kleene [Kle52]: Every continuous function w.r.t. a complete order has a least fixpoint. A general formulation of this theorem can be found in, e.g., [Gie14].

Theorem 4.1.14 (Fixpoint Theorem) *For every logic program* $\mathcal{P}$*,* $\underline{\text{trans}}_{\mathcal{P}}$ *has a least fixpoint* $\text{lfp}(\underline{\text{trans}}_{\mathcal{P}})$*. We have* $\text{lfp}(\underline{\text{trans}}_{\mathcal{P}}) = \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$.

Proof. First we show that $\bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$ is a fixpoint of $\underline{\text{trans}}_{\mathcal{P}}$. We have

$$\begin{array}{lcll} \underline{\text{trans}}_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)) & = & \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^{i+1}_{\mathcal{P}}(\varnothing) & \text{(due to continuity, Lemma 4.1.13 (b))} \\ & = & \varnothing \cup \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^{i+1}_{\mathcal{P}}(\varnothing) & \\ & = & \bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^{i}_{\mathcal{P}}(\varnothing). & \end{array}$$

Now we show that $\bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing)$ is the least fixpoint of $\underline{\text{trans}}_{\mathcal{P}}$. Let M be another fixpoint of $\underline{\text{trans}}_{\mathcal{P}}$. In order to prove $\bigcup_{i\in\mathbb{N}} \underline{\text{trans}}^i_{\mathcal{P}}(\varnothing) \subseteq M$, it suffices to show that $\underline{\text{trans}}^i_{\mathcal{P}}(\varnothing) \subseteq M$ holds for all $i \in \mathbb{N}$. We show this by induction on i. In the induction base ($i = 0$), we obviously have $\underline{\text{trans}}^0_{\mathcal{P}}(\varnothing) = \varnothing \subseteq M$.

In the induction step, we assume $\underline{\text{trans}}^{i-1}_{\mathcal{P}}(\varnothing) \subseteq M$ as induction hypothesis. Due to the monotonicity of $\underline{\text{trans}}_{\mathcal{P}}$ (Lemma 4.1.13 (a)), we have $\underline{\text{trans}}^i_{\mathcal{P}}(\varnothing) \subseteq \underline{\text{trans}}_{\mathcal{P}}(M) = M$, since M is a fixpoint of $\underline{\text{trans}}_{\mathcal{P}}$. □

Now we can define the fixpoint semantics.

Definition 4.1.15 (Fixpoint Semantics of a Logic Program) *Let* $\mathcal{P}$ *be a logic program and let* $G = \{\neg A_1, \ldots, \neg A_k\}$ *be a query. So here,* A_1*, ...,* A_k *are atomic formulas. Then the* fixpoint semantics *of* $\mathcal{P}$ *w.r.t.* G *is defined as*

$$F[\![\mathcal{P}, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid \sigma(A_i) \in \text{lfp}(\underline{\text{trans}}_{\mathcal{P}}) \text{ for all } i\}.$$

The following theorem shows that the fixpoint semantics is also equivalent to the two other forms of semantics that were introduced before.

Theorem 4.1.16 (Equivalence of Fixpoint Semantics and the Other Semantics) *Let $\mathcal{P}$ be a logic program and let $G = \{\neg A_1, \ldots, \neg A_k\}$ be a query. Then $F[\![\mathcal{P}, G]\!] = D[\![\mathcal{P}, G]\!] = P[\![\mathcal{P}, G]\!]$.*

Proof. Since $D[\![\mathcal{P}, G]\!] = P[\![\mathcal{P}, G]\!]$ holds by Theorem 4.1.8, it is enough to show $P[\![\mathcal{P}, G]\!] \subseteq F[\![\mathcal{P}, G]\!]$ and $F[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$.

Part 1: $P[\![\mathcal{P}, G]\!] \subseteq F[\![\mathcal{P}, G]\!]$

The proof is analogous to the proof of $P[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$ in Theorem 4.1.8. Let $\sigma'(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, G]\!]$. Then there is a successful computation of the form

$$(G, \varnothing) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \sigma),$$

where $\sigma'(A_1 \wedge \ldots \wedge A_k)$ is a ground instance of $\sigma(A_1 \wedge \ldots \wedge A_k)$. We have to show that then $\sigma'(A_i) \in \mathrm{lfp}(\underline{\mathrm{trans}}_{\mathcal{P}}) = \bigcup_{i \in \mathbb{N}} \underline{\mathrm{trans}}^i_{\mathcal{P}}(\varnothing)$ holds for all i. More precisely, we show that there is a $j \in \mathbb{N}$ such that $\underline{\mathrm{trans}}^j_{\mathcal{P}}(\varnothing)$ contains all ground instances of $\sigma(A_i)$ for all i.

We use induction on the length l of the computation. So the computation has the form

$$(G, \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash_{\mathcal{P}} (G_2, \delta_2 \circ \delta_1) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1)$$

with $\sigma = \delta_l \circ \ldots \circ \delta_1$. Thus, there is an A_i, a $K \in \mathcal{P}$, and a variable renaming ν with $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ and $n \geq 0$, such that G and $\nu(K)$ have no common variables such that δ_1 is the mgu of A_i and B, and such that

$$G_1 = \delta_1(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}).$$

In the induction base, we have $l = 1$. Then $G_1 = \square$ holds and therefore we have $i = k = 1$, $n = 0$, $\nu(K) = \{B\}$ (i.e., the program clause K is a fact), and $\sigma = \delta_1$. Since $\underline{\mathrm{trans}}_{\mathcal{P}}(\varnothing)$ contains all ground instances of all facts like B, $\underline{\mathrm{trans}}_{\mathcal{P}}(\varnothing)$ contains also all ground instances of $\delta_1(B) = \delta_1(A_1) = \sigma(A_1)$.

In the induction step, we have $l > 1$. Then obviously

$$(G_1, \varnothing) \vdash_{\mathcal{P}} (G_2, \delta_2) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_2)$$

is also a computation. By the induction hypothesis, there is a j such that $\underline{\mathrm{trans}}^j_{\mathcal{P}}(\varnothing)$ contains all ground instances of $\delta_l(\ldots \delta_2(\delta_1(A_p)) \ldots)$ with $p \in \{1, \ldots, k\} \backslash \{i\}$ and all ground instances of $\delta_l(\ldots \delta_2(\delta_1(C_p)) \ldots)$ with $p \in \{1, \ldots, n\}$. Since $\{B, \neg C_1, \ldots, \neg C_n\}$ was obtained from a clause of $\mathcal{P}$ by renaming, every ground instance of $\delta_l(\ldots \delta_2(\delta_1(B)) \ldots)$ is contained in $\underline{\mathrm{trans}}^{j+1}_{\mathcal{P}}(\varnothing)$. Since $\delta_1(B) = \delta_1(A_i)$ holds, $\underline{\mathrm{trans}}^{j+1}_{\mathcal{P}}(\varnothing)$ also contains all ground instances of $\delta_l(\ldots \delta_2(\delta_1(A_i)) \ldots)$.

Part 2: $F[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$

Let $\sigma(A_1 \wedge \ldots \wedge A_k) \in F[\![\mathcal{P}, G]\!]$. Then $\sigma(A_i) \in \mathrm{lfp}(\underline{\mathrm{trans}}_{\mathcal{P}})$ holds for all i. We have to shown that $\mathcal{P} \models \sigma(A_i)$ holds as well. We prove by induction on j that for all $A' \in \underline{\mathrm{trans}}^j_{\mathcal{P}}(\varnothing)$,

we have $\mathcal{P} \models A'$. Then the claim follows, since $\text{lfp}(\underline{\text{trans}}_{\mathcal{P}}) = \bigcup_{j \in \mathbb{N}} \underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$ implies that there is a $j \in \mathbb{N}$ with $\sigma(A_i) \in \underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$.

The induction base $j = 0$ is trivial, since $\underline{\text{trans}}^0_{\mathcal{P}}(\varnothing) = \varnothing$. In the induction step, let $j > 0$ and $A' \in \underline{\text{trans}}^j_{\mathcal{P}}(\varnothing)$. If already $A' \in \underline{\text{trans}}^{j-1}_{\mathcal{P}}(\varnothing)$, then the claim follows from the induction hypothesis. Otherwise, there is a clause $\{A, \neg B_1, \ldots, \neg B_n\} \in \mathcal{P}$ with a ground instance $\{A', \neg B'_1, \ldots, \neg B'_n\}$, such that we have $B'_p \in \underline{\text{trans}}^{j-1}_{\mathcal{P}}(\varnothing)$ for all $p \in \{1, \ldots, n\}$. By the induction hypothesis, we have $\mathcal{P} \models B'_p$ for all p and therefore also $\mathcal{P} \models A'$. □

4.2 Universality of Logic Programming

Now that we have defined the syntax and semantics of logic programming, we want to show that logic programming is indeed a full-fledged programming language. Thus, one can compute *any computable* function by logic programs. Such programming languages are also called "*Turing complete*".

As usual we restrict ourselves to arithmetic functions $f : \mathbb{N}^n \to \mathbb{N}$ over the natural numbers when considering the computable functions. The reason is that all other data structures can be encoded by a suitable mapping to the natural numbers.

There are different approaches to characterize the set of computable functions. One possibility is to define the set of "computable" functions as the set of functions which are computable by Turing machines. In parallel to Turing's approach, the following approach of Kleene was developed, which defines the set of computable functions as the class of μ-recursive functions. All these approaches lead to the same set of functions, i.e., the set of computable functions is always the same, no matter whether they are defined via Turing machines or via μ-recursive functions. This led to *Church's Thesis*, which states that this set is indeed the set of all intuitively computable functions.

The class of μ-recursive functions is defined inductively. One starts from certain basic functions and then one uses different principles to construct new μ–recursive functions from μ-recursive functions.

Definition 4.2.1 (μ-recursive Functions) *The class of* μ-recursive functions *is the smallest class of arithmetic functions with:*

1. *For all $n \in \mathbb{N}$, the function $\mathsf{null}_n : \mathbb{N}^n \to \mathbb{N}$ with $\mathsf{null}_n(k_1, \ldots, k_n) = 0$ is μ-recursive.*

2. *The successor function $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$ with $\mathsf{succ}(k) = k + 1$ is μ-recursive.*

3. *For all $n \geq 1$ and all $1 \leq i \leq n$, the projection function $\mathsf{proj}_{n,i}(k_1, \ldots, k_n) = k_i$ is μ-recursive.*

4. *The μ-recursive functions are closed under* composition. *So for all $m \geq 1$ and all $n \geq 0$ we have: If $f : \mathbb{N}^m \to \mathbb{N}$ and $f_1, \ldots, f_m : \mathbb{N}^n \to \mathbb{N}$ are μ-recursive, then the following function $g : \mathbb{N}^n \to \mathbb{N}$ is μ-recursive as well:*

$$g(k_1, \ldots, k_n) = f(f_1(k_1, \ldots, k_n), \ldots, f_m(k_1, \ldots, k_n))$$

5. *The μ-recursive functions are closed under* primitive recursion. *So for all $n \geq 0$ we have: If $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$ are μ-recursive, then the following function $h : \mathbb{N}^{n+1} \to \mathbb{N}$ is μ-recursive as well:*

$$\begin{array}{rcl} h(k_1, \ldots, k_n, 0) & = & f(k_1, \ldots, k_n) \\ h(k_1, \ldots, k_n, k+1) & = & g(k_1, \ldots, k_n, k, h(k_1, \ldots, k_n, k)) \end{array}$$

Thus, primitive recursion means that the definition of $h(\ldots, k+1)$ is reduced to the definition of $h(\ldots, k)$.

6. *The μ-recursive functions are closed under (unbounded)* minimization. *So for all $n \geq 0$ we have: If $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is μ-recursive, then the following function $g : \mathbb{N}^n \to \mathbb{N}$ is μ-recursive as well:*

$$g(k_1, \ldots, k_n) = k \quad \textit{iff} \quad \begin{array}{l} f(k_1, \ldots, k_n, k) = 0 \textit{ and for all } 0 \leq k' < k, \\ f(k_1, \ldots, k_n, k') \textit{ is defined and } f(k_1, \ldots, k_n, k') > 0 \end{array}$$

If there is no such k, then $g(k_1, \ldots, k_n)$ is undefined, i.e., minimization can yield partial functions.

The class of of primitive recursive *functions are those functions that are constructed using only Items 1 – 5.*

Of course, the class of primitive recursive functions does not contain all computable functions, because all primitive recursive functions are total and there are obviously non-total partial functions that are computable (by using non-terminating programs). Moreover, there are also total functions like the so-called *Ackermann* function that are computable but not primitive recursive. The following examples clarify the definition of primitive recursive and μ-recursive functions.

Example 4.2.2 *The addition function $\mathsf{plus} : \mathbb{N}^2 \to \mathbb{N}$ is primitive recursive, because it can be represented using primitive recursion as follows. Here, f is the 3-ary function with $\mathsf{f}(x, y, z) = z + 1$:*

$$\begin{array}{rcl} \mathsf{f}(x, y, z) & = & \mathsf{succ}(\mathsf{proj}_{3,3}(x, y, z)) \\ \mathsf{plus}(x, 0) & = & \mathsf{proj}_{1,1}(x) \\ \mathsf{plus}(x, y+1) & = & \mathsf{f}(x, y, \mathsf{plus}(x, y)) \end{array}$$

Similarly, the multiplication function $\mathsf{times} : \mathbb{N}^2 \to \mathbb{N}$ is primitive recursive. Here, g is the 3-ary function with $\mathsf{g}(x, y, z) = x + z$:

$$\begin{array}{rcl} \mathsf{g}(x, y, z) & = & \mathsf{plus}(\mathsf{proj}_{3,1}(x, y, z), \mathsf{proj}_{3,3}(x, y, z)) \\ \mathsf{times}(x, 0) & = & \mathsf{null}_1(x) \\ \mathsf{times}(x, y+1) & = & \mathsf{g}(x, y, \mathsf{times}(x, y)) \end{array}$$

Another primitive recursive function is the predecessor function $\mathsf{p} : \mathbb{N} \to \mathbb{N}$ with $\mathsf{p}(0) = 0$ and $\mathsf{p}(x+1) = x$:

$$\begin{array}{rcl} \mathsf{p}(0) & = & \mathsf{null}_0 \\ \mathsf{p}(x+1) & = & \mathsf{proj}_{2,1}(x, \mathsf{p}(x)) \end{array}$$

The following function $\mathsf{minus} : \mathbb{N}^2 \to \mathbb{N}$ *is also primitive recursive. We have* $\mathsf{minus}(x, y) = 0$ *if* $x \leq y$ *and* $\mathsf{minus}(x, y) = x - y$ *otherwise. (More precisely, we get* $\mathsf{minus}(x, y) = \mathsf{p}^y(x)$*.)*

$$\begin{array}{rcl} \mathsf{h}(x, y, z) & = & \mathsf{p}(\mathsf{proj}_{3,3}(x, y, z)) \\ \mathsf{minus}(x, 0) & = & \mathsf{proj}_{1,1}(x) \\ \mathsf{minus}(x, y + 1) & = & \mathsf{h}(x, y, \mathsf{minus}(x, y)) \end{array}$$

Finally, we show that the function $\mathsf{div} : \mathbb{N}^2 \to \mathbb{N}$ *is* μ*-recursive. Here* $\mathsf{div}(x, y) = \lceil \frac{x}{y} \rceil$ *and* $\mathsf{div}(0, 0) = 0$*. But* $\mathsf{div}(x + 1, 0)$ *is undefined.*

$$\mathsf{div}(x, y) = z \quad \textit{iff} \quad \begin{array}{l} \mathsf{i}(x, y, z) = 0 \textit{ and for all } 0 \leq z' < z, \mathsf{i}(x, y, z') \textit{ is defined and} \\ \mathsf{i}(x, y, z') > 0 \end{array}$$

Here, $\mathsf{i}(x, y, z) = x - (y \cdot z)$*, i.e.,*

$$\begin{array}{rcl} \mathsf{i}(x, y, z) & = & \mathsf{minus}(\mathsf{proj}_{3,1}(x, y, z), \mathsf{j}(x, y, z)) \\ \mathsf{j}(x, y, z) & = & \mathsf{times}(\mathsf{proj}_{3,2}(x, y, z), \mathsf{proj}_{3,3}(x, y, z)) \end{array}$$

Now we want to show that every computable (i.e., μ-recursive) function can be computed by a logic program. To this end, we first have to determine how functions can be computed by logic programs. The reason is that in logic programs one only defined relations instead of functions. The solution is to replace any n-ary function $f : \mathbb{N}^n \to \mathbb{N}$ by an $(n + 1)$-ary relation $\underline{\mathtt{f}} \subseteq \mathbb{N}^{n+1}$ which corresponds to the graph of f.

We also have to explain how natural numbers can be represented by terms, since logic programs only operate on terms. For this, we use a representation with the function symbols $\mathtt{0} \in \Sigma_0$ and $\mathtt{s} \in \Sigma_1$, where $\mathtt{0}$ represents the number 0, $\mathtt{s}(\mathtt{0})$ represents the number 1, $\mathtt{s}(\mathtt{s}(\mathtt{0}))$ represents the number 2, etc. Thus, the function symbol $\mathtt{s}$ represents the successor function.

Definition 4.2.3 (Computing Arithmetic Functions by Logic Programs)

- *Every number* $k \in \mathbb{N}$ *is represented by the term* $\underline{k} \in \mathcal{T}(\Sigma, \mathcal{V})$ *with* $\underline{k} = \mathtt{s}^k(\mathtt{0})$*, where* $\mathtt{0} \in \Sigma_0$ *and* $\mathtt{s} \in \Sigma_1$*.*
- *A logic program* $\mathcal{P}$ *over* (Σ, Δ) *computes an arithmetic function* $f : \mathbb{N}^n \to \mathbb{N}$ *iff there is a predicate symbol* $\underline{\mathtt{f}} \in \Delta_{n+1}$ *such that*

$$f(k_1, \ldots, k_n) = k \quad \textit{iff} \quad \mathcal{P} \models \underline{\mathtt{f}}(\underline{k_1}, \ldots, \underline{k_n}, \underline{k}).$$

The motivation for the above definition is that then the function f can be computed by queries to the logic program. To compute $f(k_1, \ldots, k_n)$, one poses the query $\mathtt{?-}\ \underline{\mathtt{f}}(\underline{k_1}, \ldots, \underline{k_n}, \mathtt{X})$. to the logic program.

Example 4.2.4 *The following logic program computes the functions from Example 4.2.2. However, the clauses below do not result directly from the transformation of the definitions from Example 4.2.2, but we have chosen alternative definitions to increase readability. (In particular, this applies to the clauses of the division predicate* $\underline{\mathtt{div}}$*).*

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

times(X,0,0).
times(X,s(Y),Z) :- times(X,Y,U), plus(X,U,Z).

p(0,0).
p(s(X),X).

minus(X,0,X).
minus(X,s(Y),Z) :- minus(X,Y,U), p(U,Z).

div(0,Y,0).
div(s(X),s(Y),s(Z)) :- minus(X,Y,U), div(U,s(Y),Z).
```

The following theorem proves the universality of logic programming.

Theorem 4.2.5 (Universality of Logic Programming) *Every μ-recursive function can be computed by a logic program.*

Proof. The proof is done by induction on the structure of the class of μ-recursive functions.

1. The function null_n can be computed by the following logic program:
   $\mathtt{null_n}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{0})$.
2. The successor function succ can be computed by the following logic program:
   $\mathtt{succ}(\mathtt{X}, \mathtt{s}(\mathtt{X}))$.
3. The projection function $\mathsf{proj}_{n,i}$ can be computed by the following logic program:
   $\mathtt{proj_{n,i}}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{X_i})$.
4. Now we show how composition can be realized by logic programs. Let $f : \mathbb{N}^m \to \mathbb{N}$ and $f_1, \ldots, f_m : \mathbb{N}^n \to \mathbb{N}$ be μ-recursive. Then by the induction hypothesis there is a logic program with predicates $\mathtt{f} \in \Delta_{m+1}$ and $\mathtt{f_1}, \ldots, \mathtt{f_m} \in \Delta_{n+1}$ that computes these functions. Let g be defined by composition of f with $f_1, \ldots, f_m$, i.e.,
   $$g(k_1, \ldots, k_n) = f(f_1(k_1, \ldots, k_n), \ldots, f_m(k_1, \ldots, k_n)).$$
   To compute g, the logic program is extended by the following clause:
   $\mathtt{g}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Z})$:- $\mathtt{f_1}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Y_1}), \ldots, \mathtt{f_m}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{Y_m}), \mathtt{f}(\mathtt{Y_1}, \ldots, \mathtt{Y_m}, \mathtt{Z})$.
5. Next we show how primitive recursion can be realized in logic programming. Let $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$ be μ-recursive. Then by the induction hypothesis, there is a logic program with predicates $\mathtt{f} \in \Delta_{n+1}$ and $\mathtt{g} \in \Delta_{n+3}$ that computes these functions. Let h be defined by primitive recursion with f and g, i.e.,
   $$\begin{aligned} h(k_1, \ldots, k_n, 0) &= f(k_1, \ldots, k_n) \\ h(k_1, \ldots, k_n, k+1) &= g(k_1, \ldots, k_n, k, h(k_1, \ldots, k_n, k)) \end{aligned}$$

To compute h, the logic program is extended by the following clauses:

$\underline{\mathtt{h}}(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{0},\mathtt{Z})$:- $\underline{\mathtt{f}}(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{Z})$.

$\underline{\mathtt{h}}(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{s}(\mathtt{X}),\mathtt{Z})$:- $\underline{\mathtt{h}}(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{X},\mathtt{Y}),\underline{\mathtt{g}}(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{X},\mathtt{Y},\mathtt{Z})$.

6. Finally, we show how the minimization can be realized by logic programs. Let $f : \mathbb{N}^{n+1} \to \mathbb{N}$ be μ-recursive. Then by the induction hypothesis, there is a logic program with a predicate $\underline{\mathtt{f}} \in \Delta_{n+2}$ that computes this function. Let g be defined by minimization with f, i.e.,

$$g(k_1,\ldots,k_n) = k \quad \text{iff} \quad \begin{array}{l} f(k_1,\ldots,k_n,k) = 0 \text{ and for all } 0 \le k' < k, \\ f(k_1,\ldots,k_n,k') \text{ is defined and } f(k_1,\ldots,k_n,k') > 0 \end{array}$$

To compute g, the logic program is extended by the following clauses. Here, $\underline{\mathtt{f}}'(X_1,\ldots,X_n,Y,Z)$ holds iff $f(X_1,\ldots,X_n,Z) = 0$ and for all X with $Y \le X < Z$, we have $f(X_1,\ldots,X_n,X) > 0$.

$\underline{\mathtt{g}}(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{Z})$:- $\underline{\mathtt{f}}'(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{0},\mathtt{Z})$.

$\underline{\mathtt{f}}'(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{Y},\mathtt{Y})$:- $\underline{\mathtt{f}}(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{Y},\mathtt{0})$.
$\underline{\mathtt{f}}'(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{Y},\mathtt{Z})$:- $\underline{\mathtt{f}}(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{Y},\mathtt{s}(\mathtt{U})),\underline{\mathtt{f}}'(\mathtt{X_1},\ldots,\mathtt{X_n},\mathtt{s}(\mathtt{Y}),\mathtt{Z})$.

□

Example 4.2.6 *The following logic program is obtained by applying the procedure from the proof of Theorem 4.2.5 to the definition of the* plus *function from Example 4.2.2.*

$\underline{\mathtt{proj_{3,3}}}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{Z})$.

$\underline{\mathtt{succ}}(\mathtt{X},\mathtt{s}(\mathtt{X}))$.

$\underline{\mathtt{f}}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{V})$:- $\underline{\mathtt{proj_{3,3}}}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{U}),\underline{\mathtt{succ}}(\mathtt{U},\mathtt{V})$.

$\underline{\mathtt{proj_{1,1}}}(\mathtt{X},\mathtt{X})$.

$\underline{\mathtt{plus}}(\mathtt{X},\mathtt{0},\mathtt{U})$:- $\underline{\mathtt{proj_{1,1}}}(\mathtt{X},\mathtt{U})$.
$\underline{\mathtt{plus}}(\mathtt{X},\mathtt{s}(\mathtt{Y}),\mathtt{U})$:- $\underline{\mathtt{plus}}(\mathtt{X},\mathtt{Y},\mathtt{Z}),\underline{\mathtt{f}}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{U})$.

If one applies the procedure from the proof to the definition of the div *function from Example 4.2.2, one obtains the following clauses:*

$\underline{\mathtt{div}}(\mathtt{X1},\mathtt{X2},\mathtt{Z})$:- $\underline{\mathtt{i}}'(\mathtt{X1},\mathtt{X2},\mathtt{0},\mathtt{Z})$.

$\underline{\mathtt{i}}'(\mathtt{X1},\mathtt{X2},\mathtt{Y},\mathtt{Y})$:- $\underline{\mathtt{i}}(\mathtt{X1},\mathtt{X2},\mathtt{Y},\mathtt{0})$.
$\underline{\mathtt{i}}'(\mathtt{X1},\mathtt{X2},\mathtt{Y},\mathtt{Z})$:- $\underline{\mathtt{i}}(\mathtt{X1},\mathtt{X2},\mathtt{Y},\mathtt{s}(\mathtt{U})),\underline{\mathtt{i}}'(\mathtt{X1},\mathtt{X2},\mathtt{s}(\mathtt{Y}),\mathtt{Z})$.

Here, we have $\underline{\mathtt{i}}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{U})$ *iff* $\mathtt{X} - (\mathtt{Y} \cdot \mathtt{Z}) = \mathtt{U}$.

4.3 Indeterminism and Evaluation Strategies

To *execute* logic programs, one proceeds as in the definition of the procedural semantics. If the query G is posed to the logic program $\mathcal{P}$, one tries to find a computation $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma)$ and then returns the answer substitution σ restricted to the variables of G.

However, the definition of $\vdash_{\mathcal{P}}$ (Def. 4.1.5) still has two indeterminisms in each step $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$:

- **Indeterminism 1**: This indeterminism is the choice of the program clause $K \in \mathcal{P}$ which is used for the resolution with G_1.
- **Indeterminism 2**: This indeterminism is the choice of the literal A_i in G_1 that is used for the resolution step.

Thus, for a configuration (G_1, σ_1) there can be *several* configurations (G_2, σ_2) with $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$. This is illustrated by the following example.

Example 4.3.1 *We consider the following variant of the logic program from Chapter 1*

```
motherOf(renate, susanne).
motherOf(susanne, aline).

ancestor(V,X) :- motherOf(V,X).
ancestor(V,X) :- motherOf(V,Y), ancestor(Y,X).
```

and the query

```
?- ancestor(Z,aline).
```

There are two possibilities for the first computation step, as one can perform resolution with two different program clauses. This corresponds to Indeterminism 1:

$$(\{\neg\mathsf{ancestor}(Z, \mathsf{aline})\}, \varnothing) \vdash_{\mathcal{P}} (\{\neg\mathsf{motherOf}(Z, \mathsf{aline})\}, \{V/Z, X/\mathsf{aline}\})$$
$$(\{\neg\mathsf{ancestor}(Z, \mathsf{aline})\}, \varnothing) \vdash_{\mathcal{P}} (\{\neg\mathsf{motherOf}(Z, Y), \neg\mathsf{ancestor}(Y, \mathsf{aline})\}, \{V/Z, X/\mathsf{aline}\})$$

If one decides on the second computation step, one obtains the query

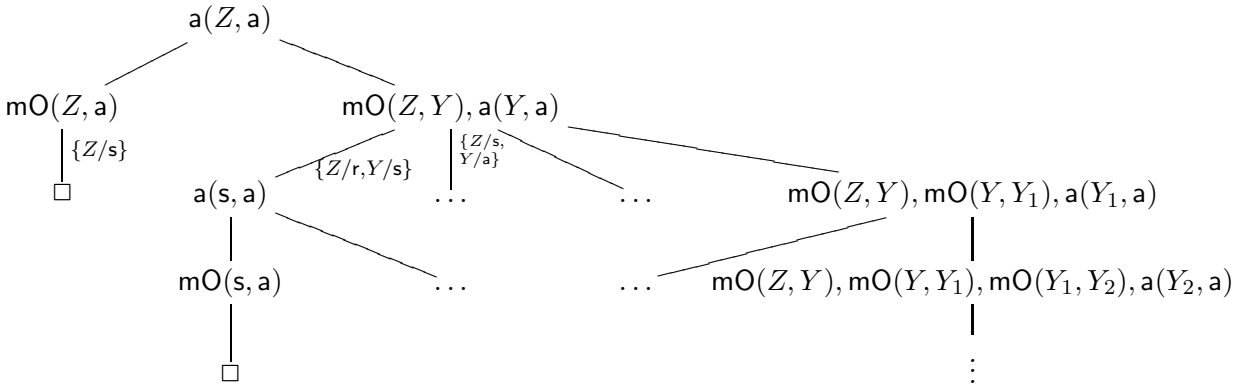
```
?- motherOf(Z,Y), ancestor(Y,aline).
```

Now one also has to deal with Indeterminism 2. If one decides to use the first literal $\neg\mathsf{motherOf}(Z, Y)$ *for resolution, then there are two possibilities to continue (because of Indeterminism 1 and the two* motherOf *facts). If one performs resolution with the clause* $\{\mathsf{motherOf}(\mathsf{susanne}, \mathsf{aline})\}$*, then one cannot obtain a successful computation anymore. On the other hand, if one performs resolution with the clause* $\{\mathsf{motherOf}(\mathsf{renate}, \mathsf{susanne})\}$*, then one can generate a successful computation which leads to the answer substitution* $\{Z/\mathsf{renate}\}$.

If one decides to use the second literal $\neg\mathsf{ancestor}(Y, \mathsf{aline})$ *for resolution instead, then (because of Indeterminism 1 and the two* ancestor *facts) there are also two possibilities to continue, etc. In particular, one can also obtain an infinite computation by always using*

the second (recursive) ancestor *clause for Indeterminism 1 and by always using the last* ancestor *literal for Indeterminism 2.*

The following tree shows the possibilities for computations. Here, instead of a pair $(\{\neg A_1, \ldots, \neg A_k\}, \sigma)$*, in the nodes we only gave the atoms* $A_1, \ldots, A_k$ *of the query. A computation step of the form* $(G_1, \sigma) \vdash_{\mathcal{P}} (G_2, \sigma' \circ \sigma)$ *is represented by an edge which is labeled with* σ' *(restricted to the variables in* G_1*). In this way, one can directly see the answer substitutions obtained for successful paths. The abbreviation "*a*" stands for "*ancestor*", "*mO*" for "*motherOf*", "*s*" for "*susanne*", "*a*" for "*aline*", and "*r*" for "*renate*".*



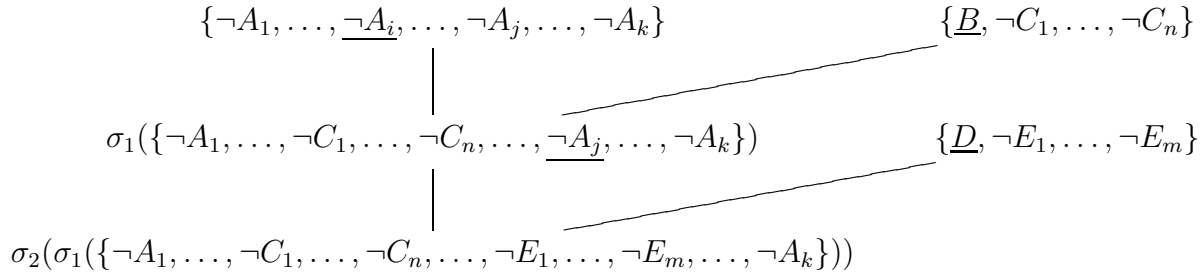
Note that Indeterminism 1 influences the solution: the leftmost successful computation yields the answer substitution $\{Z/\mathsf{susanne}\}$ *and the second successful computation results in* $\{Z/\mathsf{renate}\}$*. Also note that Indeterminism 2 influences whether one obtains an infinite or a finite computation, because the rightmost path in this tree is infinite.*

To execute logic programs on a (deterministic) computer, we have to solve these two indeterminisms. So we have to determine into which configuration (G_2, σ_2) a configuration (G_1, σ_1) should be transformed.

We first consider Indeterminism 2, i.e., the selection of the literal in the query that is used for the next resolution step. Here, we show that this indeterminism is "irrelevant", since it does not influence the result (if the computation is successful). So one does not need to look at all possibilities to select the literal, but one can solve this indeterminism in an arbitrary way without affecting the completeness of the procedure.

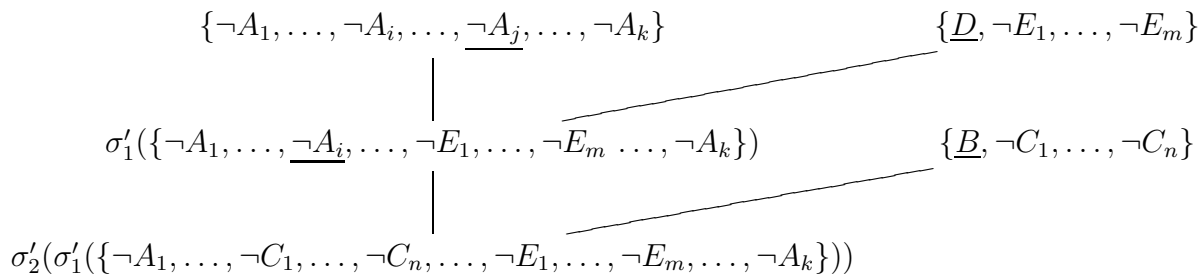
The following Exchangement Lemma is needed for this theorem and shows that one can exchange the order of the literals in the query that is used for resolution.

Lemma 4.3.2 (Exchangement Lemma) *Let* $A_1, \ldots, A_k, B, C_1, \ldots, C_n, D, E_1, \ldots, E_m \in \mathcal{A}t(\Sigma, \Delta, \mathcal{V})$*, where* $\{\neg A_1, \ldots, \neg A_k\}$*,* $\{B, \neg C_1, \ldots, \neg C_n\}$*, and* $\{D, \neg E_1, \ldots, \neg E_m\}$ *are pairwise variable-disjoint. Let* σ_1 *be the mgu of* A_i *and* B*, and let* σ_2 *be the mgu of* $\sigma_1(A_j)$ *and* D *with* $j \neq i$*. Therefore, the following (binary) SLD resolution steps are possible:*



As before, edges in this diagram are used to depict resolution steps from the parent clauses to the resolvents, and the literals used for resolution are underlined.

Then there is also an mgu σ_1' of A_j and D and an mgu σ_2' of $\sigma_1'(A_i)$ and B. Thus, the following (binary) SLD resolution steps are possible as well:



Moreover, the substitutions $\sigma_2 \circ \sigma_1$ and $\sigma_2' \circ \sigma_1'$ are the same up to variable renaming (i.e., there is a variable renaming ν such that $\sigma_2' \circ \sigma_1' = \nu \circ \sigma_2 \circ \sigma_1$).

Proof. Since the clauses are variable-disjoint, the mgu σ_1 of A_i and B does not change the variables of D, i.e., $\sigma_1(D) = D$. Then σ_2 is not only a unifier of $\sigma_1(A_j)$ and D, but also a unifier of $\sigma_1(A_j)$ and $\sigma_1(D)$. From $\sigma_2(\sigma_1(A_j)) = \sigma_2(\sigma_1(D))$ we can conclude that A_j and D are unifiable with the unifier $\sigma_2 \circ \sigma_1$, and therefore, there exists an mgu σ_1'. Hence, there is a substitution σ with

$$\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1'. \tag{4.5}$$

So it is possible to start with a resolution step on $\neg A_j$. It remains to show that afterwards a resolution step on $\sigma_1'(\neg A_i)$ and B can be done, i.e., that $\sigma_1'(A_i)$ and B are unifiable. This holds because σ is a unifier:

$$\begin{array}{rcll} \sigma(\sigma_1'(A_i)) & = & \sigma_2(\sigma_1(A_i)) & \text{by (4.5)} \\ & = & \sigma_2(\sigma_1(B)) & \text{as } \sigma_1 \text{ is unifier of } A_i \text{ and } B \\ & = & \sigma(\sigma_1'(B)) & \text{by (4.5)} \\ & = & \sigma(B) & \text{as w.l.o.g. the mgu } \sigma_1' \text{ of } A_j \text{ and } D \\ & & & \text{does not change the variables of } B \end{array}$$

Since σ is a unifier of $\sigma_1'(A_i)$ and B, there is also an mgu σ_2' and a substitution δ with

$$\sigma = \delta \circ \sigma_2'. \tag{4.6}$$

Therefore, it is possible to do the second resolution step on $\neg A_i$.

It remains to show that $\sigma_2 \circ \sigma_1$ and $\sigma_2' \circ \sigma_1'$ are equal up to variable renaming. For this, we will show that $\sigma_2 \circ \sigma_1$ is an instance of $\sigma_2' \circ \sigma_1'$, and $\sigma_2' \circ \sigma_1'$ is an instance of $\sigma_2 \circ \sigma_1$.

That $\sigma_2 \circ \sigma_1$ is an instance of $\sigma_2' \circ \sigma_1'$ follows directly from (4.5) and (4.6), since

$$\begin{array}{rcll} \sigma_2 \circ \sigma_1 & = & \sigma \circ \sigma_1' & \text{by (4.5)} \\ & = & \delta \circ \sigma_2' \circ \sigma_1' & \text{by (4.6)} \end{array}$$

For the other direction we prove analogous relations to (4.5) and (4.6) by swapping σ_1 and σ_2 with σ_1' and σ_2', respectively.

For Proposition (4.5) we used that $\sigma_2 \circ \sigma_1$ is a unifier of A_j and D. Analogously, we now use that $\sigma'_2 \circ \sigma'_1$ also unifies A_i and B. The reason is the following:

$$\begin{array}{rcll} \sigma'_2(\sigma'_1(A_i)) & = & \sigma'_2(B) & \text{since } \sigma'_2 \text{ is a unifier of } \sigma'_1(A_i) \text{ and } B \\ & = & \sigma'_2(\sigma'_1(B)) & \text{since w.l.o.g. the mgu } \sigma'_1 \text{ of } A_j \text{ and } D \\ & & & \text{does not change the variables of } B \end{array}$$

Since $\sigma'_2 \circ \sigma'_1$ is also a unifier of A_i and B, and since σ_1 is their mgu, there exists a substitution σ' such that

$$\sigma'_2 \circ \sigma'_1 = \sigma' \circ \sigma_1. \tag{4.7}$$

For Proposition (4.6) we used that σ is a unifier of $\sigma'_1(A_i)$ and B. Analogously, we now use that σ' is a unifier of $\sigma_1(A_j)$ and D:

$$\begin{array}{rcll} \sigma'(\sigma_1(A_j)) & = & \sigma'_2(\sigma'_1(A_j)) & \text{by (4.7)} \\ & = & \sigma'_2(\sigma'_1(D)) & \text{since } \sigma'_1 \text{ is a unifier of } A_j \text{ and } D \\ & = & \sigma'(\sigma_1(D)) & \text{by (4.7)} \\ & = & \sigma'(D) & \text{since w.l.o.g. the mgu } \sigma_1 \text{ of } A_i \text{ and } B \\ & & & \text{does not change the variables of } D \end{array}$$

Since σ' is a unifier of $\sigma_1(A_j)$ and D, and since σ_2 is their mgu, there is a substitution δ' with

$$\sigma' = \delta' \circ \sigma_2. \tag{4.8}$$

That $\sigma'_2(\sigma'_1(K))$ is an instance of $\sigma_2(\sigma_1(K))$ now follows immediately from (4.7) and (4.8), since

$$\begin{array}{rcll} \sigma'_2 \circ \sigma'_1 & = & \sigma' \circ \sigma_1 & \text{by (4.7)} \\ & = & \delta' \circ \sigma_2 \circ \sigma_1 & \text{by (4.8)} \end{array}$$

□

Example 4.3.3 *To illustrate the Exchangement Lemma, we consider the program*

```
p(Z,Z) :- r(Z).
q(W).
```

and the query

```
?- p(X,Y), q(X).
```

If one resolves first with the p *literal and then with the* q *literal, one gets, e.g.,*

$$\begin{array}{rcl} (\{\neg \mathsf{p}(X,Y), \neg \mathsf{q}(X)\}, \varnothing) & \vdash_{\mathcal{P}} & (\{\neg \mathsf{r}(Z), \neg \mathsf{q}(Z)\}, \{X/Z, Y/Z\}) \\ & \vdash_{\mathcal{P}} & (\{\neg \mathsf{r}(Z)\}, \{W/Z\} \circ \{X/Z, Y/Z\}) \end{array}$$

On the other hand, if one resolves first with the q *literal and then with the* p *literal, one gets, e.g.,*

$$\begin{array}{rcl} (\{\neg \mathsf{p}(X,Y), \neg \mathsf{q}(X)\}, \varnothing) & \vdash_{\mathcal{P}} & (\{\neg \mathsf{p}(W,Y)\}, \{X/W\}) \\ & \vdash_{\mathcal{P}} & (\{\neg \mathsf{r}(Y)\}, \{W/Y, Z/Y\} \circ \{X/W\}) \end{array}$$

The two resolvents and computed substitutions are therefore the same up to the variable renaming $\nu = \{Y/Z, Z/Y\}$.

The Exchangement Lemma implies that one can choose any order of the literals in the queries and impose the restriction to always perform resolution using the "first" literal according to this order. If there is a successful computation at all, then there is also a successful computation that satisfies this restriction. As mentioned at the beginning of Section 4.1, we therefore consider clauses in logic programming as *sequences* instead of *sets* of literals. So now we can choose any *selection function*, which selects the literal from the query that is used for resolution. So now the meaning of the "selection function" in the abbreviation "SLD" becomes clear.

In the programming language Prolog, one uses the selection function which always selects the first (i.e., the leftmost) literal in the query. We call this restriction of the computation sequences (with the relation $\vdash_{\mathcal{P}}$) *canonical* computations.

Definition 4.3.4 (Canonical Computation) *A computation* $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \ldots$ *of a logic program* $\mathcal{P}$ *is* canonical *iff in every resolution step one performs resolution with the leftmost literal of the respective clause* G_i*.*

To show that we can indeed restrict ourselves to canonical computations, we need the following lemma. It states that variable renamings are irrelevant in computations. So if two queries differ only by a variable renaming, then one can do the same computations with them and obtain answer substitutions which are also the same up to variable renaming.

Lemma 4.3.5 (Variable Renamings in Computations) *Let* $l \in \mathbb{N}$*. If we have* $(G, \sigma) \vdash^{l}_{\mathcal{P}} (G', \sigma')$ *and* ν *is a variable renaming, then we also have* $(\nu(G), \nu \circ \sigma) \vdash^{l}_{\mathcal{P}} (\nu(G'), \nu \circ \sigma')$*.*

Proof. We prove the lemma by induction on l. The induction base $l = 0$ is trivial, because then we have $G' = G$ and $\sigma' = \sigma$.

In the induction step $l \geq 1$ the computation is as follows:

$$(G, \sigma) \vdash_{\mathcal{P}} (H, \delta \circ \sigma) \vdash^{l-1}_{\mathcal{P}} (G', \sigma')$$

By the induction hypothesis, we have $(\nu(H), \nu \circ \delta \circ \sigma) \vdash^{l-1}_{\mathcal{P}} (\nu(G'), \nu \circ \sigma')$. Thus, it suffices to show that we also have $(\nu(G), \nu \circ \sigma) \vdash_{\mathcal{P}} (\nu(H), \nu \circ \delta \circ \sigma)$.

We have $G = \{\neg A_1, \ldots, \neg A_k\}$ and there is a program clause $K \in \mathcal{P}$ and a variable renaming ω with $\omega(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ such that G and $\omega(K)$ have no common variables. W.l.o.g. we can assume that the variables in $\omega(K)$ are all fresh and therefore $\nu(G)$ and $\omega(K)$ also have no common variables. Then δ is the mgu of an A_i and B, and the following holds

$$H = \delta(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}).$$

It is sufficient to show that $\nu \circ \delta \circ \nu^{-1}$ is an mgu of $\nu(A_i)$ and B. Then the resolution of $\nu(G)$ and $\omega(K)$ yields the clause

$$\begin{array}{ll} & \nu(\delta(\nu^{-1}(\{\nu(\neg A_1), \ldots, \nu(\neg A_{i-1}), \neg C_1, \ldots, \neg C_n, \nu(\neg A_{i+1}), \ldots, \nu(\neg A_k)\}))) \\ = & \nu(\delta(\{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\})) \\ = & \nu(H). \end{array}$$

The second step holds because ν^{-1} does not operate on the fresh variables in $\omega(K)$ and therefore $\nu^{-1}(C_j) = C_j$. So we have $(\nu(G), \nu\circ\sigma) \vdash_{\mathcal{P}} (\nu(H), \nu\circ\delta\circ\nu^{-1}\circ\nu\circ\sigma) = (\nu(H), \nu\circ\delta\circ\sigma)$.

First we show that $\nu \circ \delta \circ \nu^{-1}$ is a unifier of $\nu(A_i)$ and B. We have

$$\begin{array}{lll} & \nu(\delta(\nu^{-1}(\nu(A_i)))) & \\ = & \nu(\delta(A_i)) & \\ = & \nu(\delta(B)) & \text{since } \delta \text{ is a unifier of } A_i \text{ and } B \\ = & \nu(\delta(\nu^{-1}(B))) & \text{since } \nu^{-1} \text{ does not operate on the fresh variables in } \omega(K) \end{array}$$

Now we show that $\nu \circ \delta \circ \nu^{-1}$ is also an mgu of $\nu(A_i)$ and B. Let τ be another unifier of $\nu(A_i)$ and B. So $\tau(\nu(A_i)) = \tau(B)$ and since ν does not operate on the fresh variables in $\omega(K)$ and therefore $\nu(B) = B$, we obtain $\tau(\nu(A_i)) = \tau(\nu(B))$. So $\tau \circ \nu$ is a unifier of A_i and B and since δ is their mgu, there exists a substitution ξ with $\tau\circ\nu = \xi\circ\delta$. So we obtain $\tau \circ \nu \circ \nu^{-1} = \xi \circ \delta \circ \nu^{-1}$. This results in $\tau = \xi \circ \delta \circ \nu^{-1}$ and finally $\tau = \xi \circ \nu^{-1} \circ \nu \circ \delta \circ \nu^{-1}$. Therefore, τ is indeed an instance of $\nu \circ \delta \circ \nu^{-1}$. □

Example 4.3.6 *We illustrate the above lemma again with the program from Example 4.3.3. We have*

$$(\{\neg\mathsf{p}(X,Y), \neg\mathsf{q}(X)\}, \sigma) \vdash_{\mathcal{P}} (\{\neg\mathsf{r}(Y), \neg\mathsf{q}(Y)\}, \{X/Y, Z/Y\} \circ \sigma)$$

for all substitutions σ. Now let $\nu = \{X/Y, Y/U, U/X\}$ be a variable renaming. Then we have

$$\begin{array}{rcl} (\nu(\{\neg\mathsf{p}(X,Y), \neg\mathsf{q}(X)\}), \nu \circ \sigma) & = & (\{\neg\mathsf{p}(Y,U), \neg\mathsf{q}(Y)\}, \nu \circ \sigma) \\ & \vdash_{\mathcal{P}} & (\{\neg\mathsf{r}(U), \neg\mathsf{q}(U)\}, \{Y/U, Z/U\} \circ \nu \circ \sigma) \\ & = & (\nu(\{\neg\mathsf{r}(Y), \neg\mathsf{q}(Y)\}), \{Y/U, Z/U\} \circ \nu \circ \sigma). \end{array}$$

Here, we have

$$\begin{array}{rcl} \{Y/U, Z/U\} \circ \nu & = & \{Y/U, Z/U\} \circ \{X/Y, Y/U, U/X\} \\ & = & \{X/U, Y/U, Z/U, U/X\}. \end{array}$$

Thus, this is the same substitution as

$$\begin{array}{rcl} \nu \circ \{X/Y, Z/Y\} & = & \{X/Y, Y/U, U/X\} \circ \{X/Y, Z/Y\} \\ & = & \{X/U, Y/U, Z/U, U/X\}. \end{array}$$

Now we show the desired theorem which states that we can restrict ourselves to canonical computations. Every successful computation is still possible under this restriction. Thus, we can eliminate Indeterminism 2 and in the remainder, we will restrict ourselves to canonical computations.

Theorem 4.3.7 (Solving Indeterminism 2) *Let $\mathcal{P}$ be a logic program and let G be a query. Then for every computation $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma)$, there is a canonical computation $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma')$ of the same length, where σ and σ' are the same up to variable renaming.*

Proof. The computation is of the form

$$(G, \varnothing) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash_{\mathcal{P}} (G_2, \delta_2 \circ \delta_1) \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\square, \delta_l \circ \ldots \circ \delta_1),$$

i.e., the empty clause is obtained after l steps and $\sigma = \delta_l \circ \ldots \circ \delta_1$. Let i be the number of steps up to the first non-canonical computation step. Thus,

$$(G_i, \delta_i \circ \ldots \circ \delta_1) \vdash_{\mathcal{P}} (G_{i+1}, \delta_{i+1} \circ \ldots \circ \delta_1)$$

with $i \geq 0$ is the first non-canonical step, i.e., here we do not resolve with the leftmost literal in the respective query G_i for the first time. Here, we define $G_0 = G$ and $G_l = \square$. If the computation is already canonical, we have $i = l$. Moreover, let j be the length of the *first non-canonical* partial computation sequence. So we traverse the sequence until the first non-canonical computation step from G_i to G_{i+1}. Then j is the number of the following non-canonical computation steps until another canonical computation step follows. We prove the theorem by induction on $(l-i, j)$. Here, $l-i$ is the number of steps after the first canonical computation sequence. As the induction relation, we use the lexicographic order on numbers. So we may assume that the theorem holds for such computations of length l which either have more canonical computation steps at the beginning or which have the same number of such steps, but a shorter first non-canonical computation sequence.

If the computation is already canonical, then the theorem is trivial. Otherwise, there is an $i < l$ such that the step from G_i to G_{i+1} is the first non-canonical step. Nevertheless, there must be another canonical computation step later, because the first literal in the query must be eliminated by resolution at some point to finally obtain the empty clause $\square$. So for the length j of this first non-canonical computation sequence, we have $i + j < l$. Therefore, we obtain

$$\begin{array}{ll} & (G_i, \delta_i \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{j-1} & (G_{i+j-1}, \delta_{i+j-1} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}} & (G_{i+j}, \delta_{i+j} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}} & (G_{i+j+1}, \delta_{i+j+1} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{l-i-j-1} & (\square, \sigma). \end{array}$$

Here, the step from $(G_{i+j-1}, \delta_{i+j-1} \circ \ldots \circ \delta_1)$ to $(G_{i+j}, \delta_{i+j} \circ \ldots \circ \delta_1)$ is non-canonical and the step from $(G_{i+j}, \delta_{i+j} \circ \ldots \circ \delta_1)$ to $(G_{i+j+1}, \delta_{i+j+1} \circ \ldots \circ \delta_1)$ is canonical.

We now apply the Exchangement Lemma 4.3.2 to swap these two computation steps. This is possible because w.l.o.g. we can assume that the variables in the program clauses are renamed in such a way that they are variable-disjoint with the respective queries. Thus, according to the Exchangement Lemma there is a variable renaming ν with

$$(G_{i+j-1}, \delta_{i+j-1} \circ \ldots \circ \delta_1) \vdash_{\mathcal{P}}^2 (\nu(G_{i+j+1}), \nu \circ \delta_{i+j+1} \circ \ldots \circ \delta_1).$$

Here $\vdash_{\mathcal{P}}^2$ denotes a computation in two steps, where the first step is now canonical. Since $(G_{i+j+1}, \delta_{i+j+1} \circ \ldots \circ \delta_1) \vdash_{\mathcal{P}}^{l-i-j-1} (\square, \sigma)$ is a computation with $l-i-j-1$ steps, by Lemma 4.3.5 $(\nu(G_{i+j+1}), \nu \circ \delta_{i+j+1} \circ \ldots \circ \delta_1) \vdash_{\mathcal{P}}^{l-i-j-1} (\square, \nu \circ \sigma)$ is also a computation with $l-i-j-1$

steps. So we get the overall computation

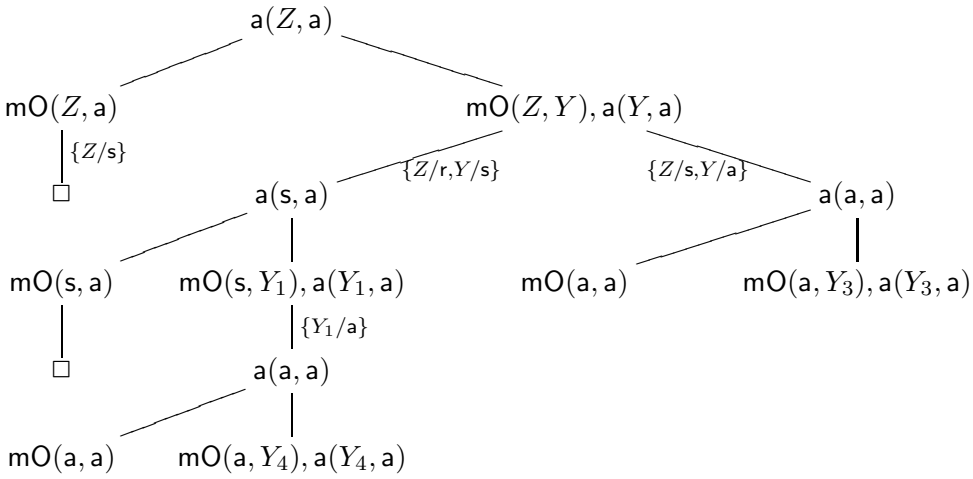
$$\begin{array}{ll} & (G, \varnothing) \\ \vdash_{\mathcal{P}}^{i} & (G_i, \delta_i \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{j-1} & (G_{i+j-1}, \delta_{i+j-1} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{2} & (\nu(G_{i+j+1}), \nu \circ \delta_{i+j+1} \circ \ldots \circ \delta_1) \\ \vdash_{\mathcal{P}}^{l-i-j-1} & (\square, \nu \circ \sigma). \end{array}$$

If $j = 1$, then the length of the first canonical computation sequence is now at least $i+1$ and thus, the induction hypothesis is applicable. If $j > 1$, then the first canonical computation sequence still has the length i, but the length of the first sequence of non-canonical steps is now only $j - 1$. So then the induction hypothesis is applicable as well.

Hence, according to the induction hypothesis there is also a canonical computation $(G, \varnothing) \vdash_{\mathcal{P}}^{+} (\square, \sigma')$, where σ' and $\nu \circ \sigma$ and thus also σ are the same up to variable renaming. □

A direct consequence of the above theorem is that completeness of SLD resolution is preserved, even we restrict ourselves to canonical computations. The same is true for any selection function which selects the literal to be used for the resolution from the negative clause.

Example 4.3.8 *In Example 4.3.1 we can now restrict ourselves to* canonical *computations and it is still ensured that we find all solutions. If we delete all non-canonical computations in the tree (i.e., if we always resolve with the first literal of the respective query), we obtain the following new tree. This tree is called* SLD tree.



In particular, note that the tree is now finite. *While queries of the form* $\{\neg\mathsf{motherOf}(\mathsf{aline}, Y_i), \neg\mathsf{ancestor}(Y_i, \mathsf{aline})\}$ *permit further (infinite) computations, they do not allow any further canonical computations, because* $\mathsf{motherOf}(\mathsf{aline}, Y_i)$ *cannot be resolved with any literal from the program clauses.*

As indicated by the previous example, Indeterminism 2 can of course influence the termination behavior of logic programs. This becomes even clearer in the following example.

Example 4.3.9 *We consider the following program.*

```
p :- p
q(a).
```

The query

```
?- q(b),p.
```

terminates in Prolog*, since there are no canonical computation steps starting from* $(\{\neg\mathsf{q}(\mathsf{b}), \neg\mathsf{p}\}, \varnothing)$*. Therefore the SLD tree only contains one node, which is labeled with the query. But there is an infinite non-canonical computation*

$$(\{\neg\mathsf{q}(\mathsf{b}), \neg\mathsf{p}\}, \varnothing) \vdash_{\mathcal{P}} (\{\neg\mathsf{q}(\mathsf{b}), \neg\mathsf{p}\}, \varnothing) \vdash_{\mathcal{P}} \ldots$$

Thus, in a programming language that always chooses the rightmost *literal of the query for resolution, this query would not terminate.*

The following definition introduces the notion *SLD tree* formally.

Definition 4.3.10 (SLD Tree) *Let $\mathcal{P}$ be a logic program and let G be a query. The* SLD tree *of $\mathcal{P}$ w.r.t. the query G is a finite or infinite tree whose nodes are labeled with sequences of atomic formulas and whose edges are labeled with substitutions. The SLD tree is the smallest tree with:*

- *If $G = \{\neg A_1, \ldots, \neg A_k\}$, then the root of the tree is labeled with $A_1, \ldots, A_k$.*
- *Now let a node be labeled with $B_1, \ldots, B_n$ and let B_1 be unifiable with the positive literals of k program clauses $K_1, \ldots, K_k$, where the clauses appear in this order in the program. Then the node has k children. The i-th child is labeled with the atoms that result from a canonical computation step by resolution with the clause K_i. If this computation has the form $(\{\neg B_1, \ldots, \neg B_n\}, \varnothing) \vdash_{\mathcal{P}} (\{\neg C_1, \ldots, \neg C_m\}, \sigma)$, then the i-th child is labeled with $C_1, \ldots, C_m$ and the edge to this child is labeled with the substitution σ restricted to the variables in $B_1, \ldots, B_n$.* [3]

Here, computation steps which result from resolution using the same literal and the same program clause, and which only differ by variable renaming are of course not distinguished. So if B_1 is unifiable with the positive literals of k program clauses, the node with the label $B_1, \ldots, B_n$ has exactly k children.

The answer substitutions can now be extracted from those paths that end with the lead $\square$. If the edges from the root to a leaf $\square$ are labeled with $\delta_1, \ldots, \delta_l$, then the resulting answer substitution is $\delta_l \circ \ldots \circ \delta_1$, restricted to the variables from the original query G. Theorem 4.3.7 implies that by regarding the SLD tree one still obtains all answer substitutions (up to variable renaming).

Besides these paths that represent successful computations, there are also paths from the root to leaves which are not the empty clause. These paths represent a *finite failure*,

[3] Of course, the labels can also differ by variable renamings.

because they end with a query whose first literal cannot be resolved with any atom of a program clause. Therefore, this query cannot lead to a successful computation anymore.

Finally, there can also be infinite paths that represent infinite computations.

In the above definition of the SLD tree, we have not only solved Indeterminism 2, but we have also considered the order of the clauses in the program to handle Indeterminism 1. The order of the children of a node now corresponds to the order of the clauses in the program.[4]

Now an *evaluation strategy* is a procedure that specifies how an SLD tree is traversed (or constructed). Such a strategy then solves Indeterminism 1. Here, one can still distinguish whether one is only interested in *one* successful computation (then the procedure stops once it has found $\square$) or whether one is interested in *all* successful computations resp. answer substitutions. In Prolog, at first only the first solution is computed. As already mentioned, the traversal of the SLD tree can be continued by entering "`;`" afterwards.

The evaluation strategy of *breadth-first search* is a complete strategy. This strategy first computes all nodes of depth 0, then all nodes of depth 1, etc. Therefore it finds all leaves $\square$ in the SLD tree and because of the completeness of canonical SLD resolution, every successful computation (i.e., every answer substitution) is eventually found in this way. If one is only interested in *one* successful computation and the original query is entailed by the program, then the procedure will terminate, because one stops as soon as the first leaf $\square$ is found. However, the procedure does not terminate if the tree is infinite and does not contain the empty clause. (So it is only semi-decidable whether the SLD tree actually has a leaf $\square$.) If one is interested in *all* successful computations, one has to construct the tree completely and thus, the procedure only terminates if the SLD tree is finite.

The disadvantage of the breadth-first search is that it is very inefficient (both in time and space). Therefore, the programming language Prolog uses *depth-first search* as its evaluation strategy instead. Here, one descends recursively into the subtrees. If one is only interested in *one* successful computation, a return to the parent nodes (backtracking) only takes place if one has found a (finite) unsuccessful computation (i.e., a leaf different from $\square$). However, due to the possibly infinite paths in the SLD tree, the depth-first search strategy is incomplete. Not every successful computation is found by depth-first search and it may even happen that the depth-first search does not find any successful computation at all, although there is one.

Example 4.3.11 *We again regard the program from Example 4.3.1 resp. Example 4.3.8. However, in the program clauses*

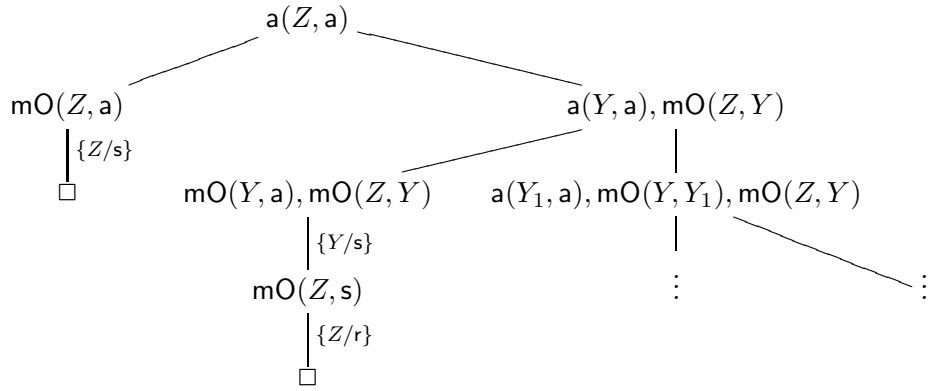
```
ancestor(V,X) :- motherOf(V,X).
ancestor(V,X) :- motherOf(V,Y), ancestor(Y,X).
```

we now replace the second clause by the following modification, where we swap the two literals in the body.

```
ancestor(V,X) :-  ancestor(Y,X), motherOf(V,Y).
```

[4] In the literature, there is often an alternative definition of the *SLD tree* where the order of the children is not fixed and therefore the order of the program clauses does not yet play a role.

This results in the following SLD tree:



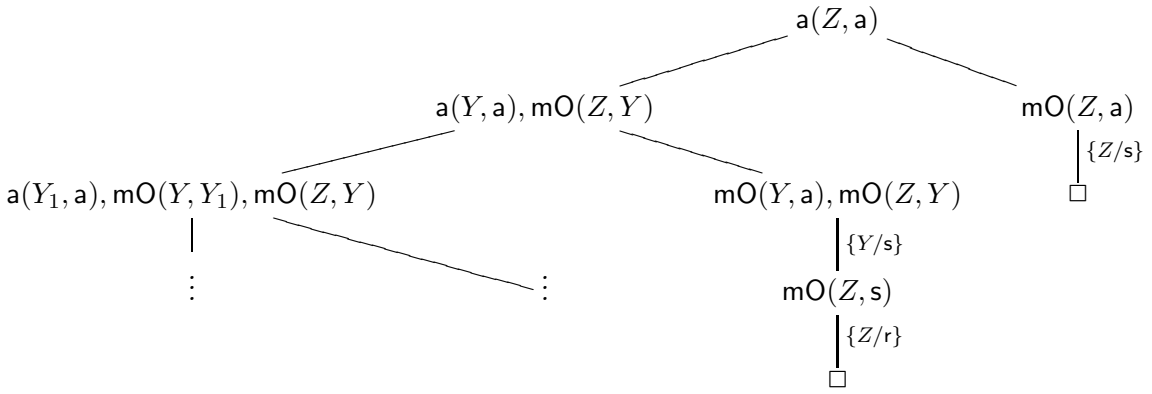
The rightmost path in this tree is now infinite. So if one uses a depth-first search which always proceeds from right *to* left*, this strategy will not find a solution in this example. However, the depth-first search used in Prolog proceeds from left to right and therefore, it finds the first two solutions. If one would search for another solution afterwards, it would not terminate anymore.*

The above example also illustrates the effect of Indeterminism 2 on the termination behavior: Swapping the literals in the body of a program clause can influence whether the SLD tree is finite and thus, it can also influence the termination of the evaluation. The following example shows the influence of Indeterminism 1.

Example 4.3.12 *We continue the above example by now also swapping the order of the clauses.*

```
ancestor(V,X) :- ancestor(Y,X), motherOf(V,Y).
ancestor(V,X) :- motherOf(V,X).
```

This results in the following SLD tree.



Now the depth-first search in Prolog would not terminate and would not find a solution.

Thus, the example illustrates that due to the evaluation strategy of Prolog, facts (resp. non-recursive clauses) should always come before the recursive rules of the same predicate.

To sum up, the evaluation strategy of Prolog has the following characteristics:

- In Prolog, literals in queries are processed *from left to right*. This way of solving Indeterminism 2 influences the termination behavior, but not the completeness of the SLD tree.
- In Prolog, program clauses are processed *from top to bottom*. This corresponds to a traversal of the SLD tree by depth-first search, where left children are always considered first.

Chapter 5

The Programming Language Prolog

Now that we have introduced the basics of logic programming, we want to consider a concrete programming language based on this principle. The best known logic programming language is Prolog. This language was developed by Kowalski and Colmerauer in the first half of the 1970s. It became one of the essential languages of AI, in particular since Prolog was chosen as the main language for the Japanese *Fifth Generation Project* in 1981.

The syntax of (simple) Prolog programs corresponds exactly to the syntax of logic programs defined in Def. 4.1.1. Here, the notation with "`:-`" for rules and with "`?-`" for queries is used. The signature of a Prolog program results from the occurring function and predicate symbols, which must begin with lower case letters. In addition, strings of special characters are allowed (e.g., `<-->`) and strings in single quotes (e.g., `'X'`). Variables begin with upper case letters or an underscore. The *anonymous* variable "`_`" plays a special role. Multiple occurrences of this variable are considered to be different and in answer substitutions, the assignment of this variable is not included. So for a program with the fact "`p(a,b,c).`", the query "`?- p(_,_,X).`" would therefore yield the answer substitution `X = c`.

Prolog allows overloading of function and predicate symbols. The above program with the fact "`p(a,b,c).`" could thus be extended, e.g., by the fact "`p(a,b).`". Here, `p` is another 2-ary predicate symbol, which has nothing to do with the 3-ary predicate symbol `p`. Similarly, the above program could be extended by the fact "`p(p(a,b),c,c)`". Now the inner `p` is a 2-ary function symbol, which is independent of the outer 3-ary predicate symbol `p`. Since symbols with different arities are considered to be different, one often mentions their arity together with their name, separated by a slash (i.e., `p/2` and `p/3`).

The semantics of Prolog corresponds to the semantics of logic programs from the previous chapter. So the SLD tree is traversed in depth-first search. Prolog stops when it has found the first answer substitution. If the user then enters a "`;`", the SLD tree is traversed further until one reaches the next occurrence of the empty clause, etc. As shown in the previous chapter, this does not correspond to a purely declarative programming paradigm, where the programmer only specifies the problem but not the procedure to find a solution. The reason is that the programmer has to understand Prologs evaluation strategy and should therefore for example write facts before recursive clauses, such that Prolog indeed finds solutions. Thus, the logic of the program is not completely separated from the control flow when executing the program.

However, an important difference to the semantics of logic programs from the previous

chapter is that most Prolog implementations omit the *occur check* when performing unification. So if a variable X has to be unified with a subterm $t \neq X$, then Prolog does not check whether X occurs in the term t. Instead, a reference to the memory cell corresponding to the term t is written to the memory cell corresponding to the variable X. Thus, if one wants to unify X with the term $\mathtt{f}(X)$, one obtains a substitution where every occurrence of X is mapped to the the term $\mathtt{f}(X)$. Thus, the resulting unifier is the substitution $\{X/\mathtt{f}(\mathtt{f}(\mathtt{f}(\ldots))\}$. Hence, the variable X is assigned the *infinite* term consisting of infinitely many `f` symbols. Such an answer substitution is obtained, e.g., for the program with the fact "`p(X,f(X)).`" and the query "`?- p(X,X).`". Thus, Prolog also has techniques to work with infinite terms (where these infinite terms are represented by finite cyclic graphs, i.e., Prolog only handles the subclass of the so-called *rational* terms).

Prolog has numerous *pre-defined* predicates, some of which are introduced in the following. In particular, there is also a pre-defined predicate `unify_with_occurs_check` which performs a correct unification (with occur check). Thus, the query

```
?- unify_with_occurs_check(X,f(X)).
```

yields the answer `false`. On the other hand, the query

```
?- unify_with_occurs_check(X,f(Y)).
```

returns the answer substitution `X = f(Y)`.

Since logic programs only work on terms as data, data objects must be represented as terms using suitable function symbols. For certain data structures (in particular for integers and for (linear) lists), Prolog provides specific notations as well as support to improve the efficiency and readability of the corresponding programs. The handling of arithmetic and lists in Prolog is shown in Sections 5.1 and 5.2. Afterwards, Section 5.3 explains how the user can define additional operators besides the built-in arithmetic operators (i.e., function symbols with infix, prefix, or postfix notation). In Section 5.4 we introduce the pre-defined *cut* predicate, which is used to cut away parts of the SLD tree and we show how to use it in order to implement *negation*. Section 5.5 explains the input and output handling in Prolog. In Section 5.6, we will demonstrate how Prolog itself can be used to manipulate Prolog programs. Finally, Section 5.7 shows how to easily implement parsers for context-free languages in Prolog.

5.1 Arithmetic

As in Definition 4.2.3, one can represent natural numbers as terms over the function symbols $\mathtt{0} \in \Sigma_0$ and $\mathtt{s} \in \Sigma_1$. Some logic programs operating on this data structure have already been presented in Example 4.2.4. For example, we had the following program for addition. (We use the function symbol `add`, since `plus` is already pre-defined in Prolog).

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

Here, "`add(X,Y,Z)`" represents the statement "`X + Y = Z`". So if one poses a query where the first two arguments of `add` are given, then this program computes addition. For example, the query "`?- add(s(0),s(s(0)),X).`" results in the answer `X = s(s(s(0)))`.

But as there are no fixed input and output arguments in logic programs, one can use this program for subtraction as well, e.g., by providing the second and the third argument. This behavior is also called *bidirectionality*. To compute "3-2", one poses the query "`?- add(X,s(s(0)),s(s(0))))`." and gets the answer `X = s(0)`.

Similarly, a query like "`?- add(X,Y,s(s(0))))`." can compute all pairs of summands for a number. The query "`?- add(X,s(s(0)),Z)`" also returns a reasonable result (all pairs of numbers where the first number is 2 less than the second, i.e., `Z = s(s(X))`). On the other hand, for the query "`?- add(s(0),Y,Z)`" there are infinitely many answer substitutions. Thus, for almost every Prolog program there are also queries which lead to non-termination or to an infinite SLD tree.

However, a disadvantage of representing natural numbers as terms over `0` and `s` is that (because of the missing built-in arithmetic operations and because of the size of the terms) this often leads to inefficient programs that are hard to understand. Therefore, Prolog allows the standard notation for integers and provides some basic functions and pre-defined built-in predicates.

An *arithmetic expression* is a term which is inductively constructed from numbers, variables, and binary infix functions like `+`, `-`, `*`, `//` (integer division), `**` (for exponentiation), etc., and the negation `-` of arity one. These expressions can be considered as terms and they can be handled by normal syntactic unification. So if one has a program with the fact "`equal(X,X).`", then the query "`?- equal(3,1+2).`" leads to the result `false`. The query "`?- equal(X,1+2).`" returns the solution `X = 1+2`.

While only the normal syntactical unification is used for the standard evaluation of queries, there are further pre-defined predicates, which use corresponding pre-defined implementations of the functions `+`, `-`, `*`, `//`, `**`, etc.

For the *comparison* of arithmetic expressions, Prolog offers the infix–predicates *op* with *op* $\in$ { `<`, `>`, `=<`, `>=`, `=:=`, `=\=` } of arity two (among others). Here the last two operators stand for equality and non-equality. A query "`?-` t_1 *op* t_2`.`" succeeds if t_1 and t_2 are fully instantiated arithmetic expressions at the time of evaluation (i.e., they may no longer contain variables) and if after the evaluation of the pre-defined infix function symbols, the values z_1 and z_2 of t_1 and t_2 are in the relation *op*. Thus, the above predicate symbols enforce an evaluation of the occurring built-in arithmetic infix functions. If t_1 or t_2 are not fully instantiated arithmetic expressions, then the query does not fail with failure, but it leads to an abort of the program. So the following queries have the following results:

- "`?- 1 < 2.`" or "`?- -2 < -1.`" or "`?- 1*1 < 1+1.`" result in `true`.
- "`?- 2 < 1.`" or "`?- 6//3 < 5-4.`" result in `false`.
- "`?- a < 1.`" or "`?- X < 1.`" lead to a program error.

The requirement that all variables must be instantiated at the time of evaluation that these predicate symbols cannot be used to instantiate variables by unification. A query like "`?- X =:= 2.`" does not lead to the answer substitution `X = 2`, but to a program error.

For that reason, there is another pre-defined predicate symbol `is`. A query "`?-` t_1 `is` t_2`.`" succeeds if t_2 is a fully instantiated arithmetic expression that evaluates to a value z_2, and if t_1 can be unified with z_2. If t_2 is not a fully instantiated arithmetic expression, the query does not fail with failure, but it again leads to an abort of the program. So the following queries have the following results:

- "`?- 2 is 1+1.`" or "`?- 2 is 2.`" result in `true`.
- "`?- 1+1 is 2.`" or "`?- 1+1 is 1+1.`" or "`?- X+1 is 1+1.`" result in `false`.
- "`?- X is 2.`" or "`?- X is 1+1.`" lead to the answer substitution `X = 2`.
- "`?- X is 3+4, Y is X+1.`" leads to the answer substitution `X = 7` and `Y = 8` (because at the time of evaluation, the `X` in "`Y is X+1`" is a fully instantiated arithmetic expression).
- "`?- X is X.`", "`?- 2 is X.`", "`?- X is a.`" or "`?- Y is X+1, X is 3+4.`" lead to a program error.

There is also a pre-defined predicate symbol `=` for the unification of arbitrary terms. This symbol is treated as if it were defined by the fact "`X = X.`" Thus, it is not restricted to arithmetic expressions. In contrast to the special symbols for arithmetic above, here the function symbols `+`, `-`, `*`, `//`, `**`, etc. are note evaluated (the same holds for other user-defined predicate symbols). So the following queries have the following results:

- "`?- a = a.`" or "`?- 2 = 2.`" or "`?- 1+1 = 1+1.`" result in `true`.
- "`?- 2 = 1+1.`" or "`?- 1+1 = 2.`" result in `false`.
- "`?- X+1 = 1+1.`" or "`?- 1 = X.`" lead to the answer substitution `X = 1`.
- "`?- X = 1+1.`" leads to the answer substitution `X = 1+1`.
- "`?- X = X.`" results in `true`, i.e., in the identical (empty) answer substitution.
- "`?- 1+X = Y+1.`" leads to the answer substitution `X = 1` and `Y = 1`.
- "`?- X = 3+4, Y is X+1.`" leads to the answer substitution `X = 3+4` and `Y = 8`.

Thus, we have different kinds of equality:

- *equality of values* "t_1 `=:=` t_2", where t_1 and t_2 are evaluated and no unification takes place.
- *value assignment* "t_1 `is` t_2", where t_2 is evaluated and then unification is used.
- *equality of terms* "t_1 `=` t_2", where no evaluation takes place, but only unification is used.
- *syntactic equality* "t_1 `==` t_2", which only examines whether t_1 and t_2 are syntactically identical.

For example, `X == X` leads to the result `true`, but `X == Y` yields the result `false`. Analogously, `f(a,X) == f(a,X)` leads to the result `true`, but `f(a,X) == f(a,Y)` yields `false`.

The following example shows how the addition program from the beginning of the section can be implemented using the pre-defined arithmetic functions. (Alternatively, one can of course also use the program "`add(X,Y,Z) :- Z is X+Y.`").

```
add(X,0,X).
add(X,Y,Z) :- Y > 0, Y1 is Y-1, add(X,Y1,Z1), Z is Z1+1.
```

As expected, the query "`?- add(1,2,X).`" leads to the answer substitution `X = 3`. The advantage of using the pre-defined functions and predicates on numbers is the improved efficiency and readability. A disadvantage is that bidirectionality can be lost. Indeed, the query "`?- add(X,2,3).`" now leads to a program error (because in the query "`Z is Z1+1`", the term `Z1` is not fully instantiated).

Note that the program

```
add(X,0,X).
add(X,Y+1,Z+1) :- add(X,Y,Z).
```

would not work as expected. The query "`?- add(1,2,X).`" leads to `false`, since `2` is neither unified with `0` nor with `Y+1`. The query "`?- add(1,0+1,X).`" results in the answer substitution `X = 1+1`.

We now present two further typical examples, i.e., the computation of the factorial and an algorithm for computing the greatest common divisor of two natural numbers.

```
fac(0,1).
fac(X,Y) :- X > 0, X1 is X-1, fac(X1,Y1), Y is X*Y1.

gcd(X,0,X).
gcd(0,X,X).
gcd(X,Y,Z) :- X =< Y, X > 0, Y1 is Y-X, gcd(X,Y1,Z).
gcd(X,Y,Z) :- Y < X, Y > 0, X1 is X-Y, gcd(X1,Y,Z).
```

The query "`?- fac(3,X).`" results in the answer substitution `X = 6` and the query "`?- gcd(28,36,X).`" results in the answer substitution `X = 4`.

Prologalso has pre-defined predicates to check the types of terms, e.g., `number/1`. Here, `number(t)` is true, if at this point, `t` is a number (i.e., the queries "`?- number(2)`" or "`?- X is 1+1, number(X).`" lead to `true`, while "`?- number(1+1).`" or "`?- number(X).`" result in `false`).

5.2 Lists

To represent lists by terms, one typically uses a function symbol of arity zero for the empty list and a function symbol of arity two that stands for the insertion of an element at the beginning of the list. If these symbols are called `nil` $\in \Sigma_0$ and `cons` $\in \Sigma_2$, the following Prolog program computes the length of a list (the predicate `length/2` is pre-defined).

```
len(nil,0).
len(cons(X,Xs),Y) :- len(Xs,Y1), Y is Y1+1.
```

Thus, the query "`?- len(cons(7,cons(3,nil)), X).`" results in the answer substitution `X = 2`.

If instead of `nil` the function symbol `[]` $\in \Sigma_0$ and instead of `cons` the function symbol[1] `'[|]'` $\in \Sigma_2$ is chosen, then Prolog supports more readable shorthand notations. As before, one can now write the following algorithm:

```
len([],0).
len('[|]'(X,Xs),Y) :- len(Xs,Y1), Y is Y1+1.
```

However, the following shorthand notations are also possible:

- `'[|]'(`t_1`, `t_2`) = [`t_1`|`t_2`]`
- `'[|]'(`t_1`, []) = [`t_1`]`
- `'[|]'(`t_1`, '[|]'(`t_2`, '[|]'(`t_3`, `t`))) = [`t_1`, `t_2`, `t_3`|`t`]`
- `'[|]'(`t_1`, '[|]'(`t_2`, '[|]'(`t_3`, []))) = [`t_1`,`t_2`,`t_3`] = [`t_1`,`t_2`|[`t_3`|[]]] = [`t_1`|[`t_2`,`t_3`|[]]]`, etc.

These shorthand notations are considered to be *identical* to the notation with `'[|]'` and `[]`. For example, we have:

- The queries "`?- [1,2] = [1|[2]].`" or "`?- [1,2] = '[|]'(1,[2])`" or "`?- [1,2,3] = [1|[2,3|[]]].`" or "`?- '[|]'(1,'[|]'(2,[3])) = [1,2,3]`" or "`?- '[|]'(1,2) = [1|2]`" result in `true`.
- "`?- '[|]'(1,X) = [1,2,3]`" results in the answer substitution `X = [2,3]`.
- "`?- [X,[1|X]] = [[2],Y].`" results in the answer substitution `X = [2], Y = [1,2]`.

Note that `'[|]'` is really an ordinary function symbol of arity two and therefore, it can also be used to represent binary trees. Thus,

```
'[|]'('[|]'(1,2),'[|]'(3,4))
```

is actually a binary tree, which can also be represented as `[[1|2]|[3|4]]`.

The following example program checks whether an element is included in a list.

```
member(X,[X|_]).
member(X,[_|Ys]) :- member(X,Ys).
```

[1] In earlier versions of Prolog, one used the function symbol `.` $\in \Sigma_2$. However, since Version 7, SWI Prolog uses the symbol `'[|]'` instead. By calling SWI Prolog with the flag `--traditional`, it is also possible to still use the notation with ".".

The query "`?- member(X,[[a,b],1,[]]).`" returns the answer substitution `X = [a,b]`. If one searches for further solutions by typing ";", one will get `X = 1` and `X = []`. (Another input of ";" then results in `false`.) The query "`?- member(b,X).`" searches for all lists containing `b`. Hence, one obtains the infinitely many solutions `X = [b|Xs]` (all lists with the first element `b`), `X = [Y,b|Xs]` (all lists with the second element `b`), etc.

Another classical program is the following predicate `app` for concatenating lists (`append/3` is pre-defined).

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

The query "`?- app([1,2],[3,4,5],Xs).`" results in the answer substitution `Xs = [1,2,3,4,5]`. The query "`?- app(Xs,Ys,[1,2,3]).`" yields the answer substitution `Xs = []`, `Ys=[1,2,3]` and by repeated input of ";" one obtains the three other solutions. On the other hand, the query "`?- app(Xs,[],Zs).`" has infinitely many solutions again.

5.3 Operators

The standard notation for *terms* and *atoms* in Prolog is prefix notation, where the arguments are given in brackets. For example, one writes `p(X,f(a))` for predicate or function symbols `p/2`, `f/1`, and `a/0`. Such predicate and function symbols are called *functors*.

Instead, it is also possible to use predicate or function symbols of arity two in infix notation and predicate or function symbols of arity one in prefix or postfix notation (without brackets for the arguments). For this, the corresponding symbols have to be declared as *operators*. The advantage of this notation is a more user-friendly syntax with better readability. Thus, one gets closer to the goal of "programming in natural language".

For example, `+` is already pre-defined as an operator. Thus, one may write the term `2+3`. Prolog converts it into the term `+(2,3)`, i.e., the query "`?- 2+3 = +(2,3).`" results in `true`.

To define operators, so-called *directives* of the form

`:- op(`*precedence*`,` *type*`,` *name(s)*`).`

are used. It is a clause with an empty head and the system predicate `op`. The operators declared in the directive can then be used in the program after the point of their declaration. Directives are queries which are processed when loading the program. So when loading the directive, one immediately tries to prove it with the previously loaded program. For the operators `+`, `-`, and `*`, the following pre-defined directives exist:

```
:- op(500,yfx,[+,-]).
:- op(400,yfx,*).
```

The last argument of `op` contains the symbol resp. the list of symbols which are declared as operators. The *precedences* express the strength of the function symbols' binding. Thus, `*` binds more strongly than `+` and this is expressed by the smaller precedence of `*`. (So a *smaller* precedence means that the symbols binds *stronger*.)

The *type* determines the order of operator and argument. Here `f` stands for the operator and `y` and `x` stand for the arguments. For infix symbols, there are the types `xfx`, `yfx`, and `xfy`. The types for prefix symbols are `fx` and `fy`, and for postfix symbols, there are the types `xf` and `yf`.

Here the precedence of the `x` arguments must be *strictly less* than the precedence of the operator `f`. The precedence of the `y` arguments must be *less or equal* than the precedence of `f`. The precedence of an argument is the precedence of the leading operator (and the precedence of functors and of arguments in brackets is 0).

Thus, for the type `yfx`, only the left argument may have the same precedence as the operator. This means that `1+2+3` stands for `(1+2)+3`. Thus the query "`?- 1+2+3 = (1+2)+3.`" returns the answer `true` and the query "`?- 1+2+3 = 1+(2+3).`" results in `false`. In other words, such operators are *left-associative*. Hence, `5-4-3` stands for `(5-4)-3`. The query "`?- X is 5-4-3`" therefore returns the answer substitution `X = -2`. Similarly, `xfy` declares right-associative operators and `xfx` declares operators without any such association.

The term `1+2*3+4` stands for `(1+(2*3))+4`. The reason is that any operator may only have arguments with equal or lower precedence. But as `*` has a lower precedence than `+`, the two `+` terms cannot be the arguments of `*`.

Of course, operators may also be overloaded. For instance, there is an additional predefined operator `-` of arity one.

```
:- op(200,fy,-).
```

Thus, the expression `-2-3` stands for `(-2)-3`.

The following example shows how programmers can define their own operators to obtain a simple form of word processing. We use the English verb "`was`" in infix notation. This should have no association, because sentences of the kind "`laura was young was beautiful`" do not make sense. Moreover, we use the word "`of`" in infix notation, where `of` should associate to the right. Thus, "`secretary of son of john`" stands for "`secretary of (son of john)`". To ensure that `of` binds stronger than `was`, `of` should have a lower precedence. Then "`laura was secretary of john`" stands for "`laura was (secretary of john)`". Finally, we introduce the word "`the`". This is a prefix operator without association (because "`the the son`" does not make sense). Its precedence should be lower than the precedence of "`of`". Then "`the secretary of the son`" stands for "`(the secretary) of (the son)`".

Now we can write the following Prolog program:

```
:- op(300,xfx,was).
:- op(250,xfy,of).
:- op(200,fx,the).

laura was the secretary of the head of the department.
```

The fact "`laura was the secretary of the head of the department.`" therefore stands for the following atomic formula with the predicate symbol `was` and function symbols `of` and `the`.

```
was(laura,of(the(secretary),of(the(head),the(department)))).
```

Now we can pose the following query:

```
?-  Who was the secretary of the head of the department.
Who = laura

?- laura was What.
What = the secretary of the head of the department

?- Who was the secretary of the head of What.
Who = laura
What = the department
```

5.4 The Cut Predicate and Negation

In Section 5.4.1 we introduce the cut predicate which allows to control the backtracking mechanism of Prolog. This allows us to implement meta predicates like *negation*, as shown in Section 5.4.2.

5.4.1 The Cut Predicate

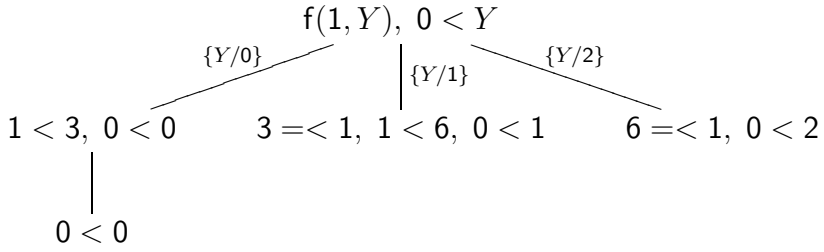
Prolog automatically performs backtracking when it reaches a leaf of the SLD tree where a finite failure has been detected (i.e., a leaf that is not the empty clause $\square$). While this is often an advantage, there are also cases where one wants to avoid backtracking. The reason is that the automatic backtracking in case of failure is both time and memory consuming, because all nodes with further children must be stored during the proof. Another reason is that backtracking and further traversal of the SLD tree can lead to unnecessary non-termination if some branches of the tree are infinite.

Consider the following simple program for computing the function f with

$$f(x) = \begin{cases} 0, \text{if } x < 3 \\ 1, \text{if } 3 \leq x < 6 \\ 2, \text{if } 6 \leq x \end{cases}$$

```
f(X,0) :- X<3.
f(X,1) :- 3=<X, X<6.
f(X,2) :- 6=<X.
```

We now regard the query "`?- f(1,Y), 0<Y.`". This leads to the following SLD tree.



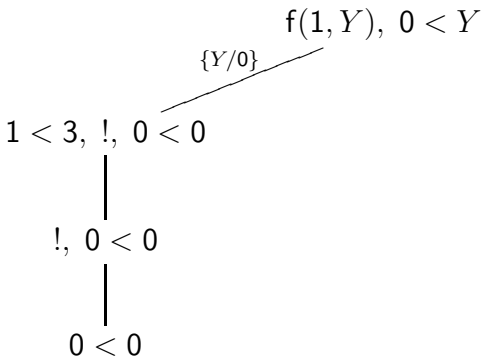
So the query fails and the result is `false`. As usual, the leftmost path of the SLD tree is traversed first. Here, the proof of the first goal `X<3` succeeds, because `X` is instantiated by `1`, but the second goal `0<Y` fails due to the instantiation of `Y` with `0`. Afterwards, one backtracks and traverses the other two paths, which correspond to the second and third program clauses.

But note that the conditions "`X<3`", "`3=<X, X<6`", and "`6=<X`" of the three `f` clauses are mutually exclusive. So as soon as the proof of one of these three conditions succeeds, there is no need to consider the other `f` clauses anymore. Since the condition `X<3` of the first `f` clause has already been proven in our query, there is no need to backtrack and consider the other two `f` clauses, since it is clear that their conditions cannot hold. We therefore want to *cut* off the middle and the right path of the SLD tree.

To this end, Prolog offers the so-called *cut* predicate, which is denoted by "`!`". This predicate symbol of arity zero can be used on the right-hand side of rules and in queries. Its proof always succeeds, but as a side effect it cuts off alternative paths in the SLD tree. To illustrate this, we now modify our program as follows:

```
f(X,0) :- X<3, !.
f(X,1) :- 3=<X, X<6, !
f(X,2) :- 6=<X.
```

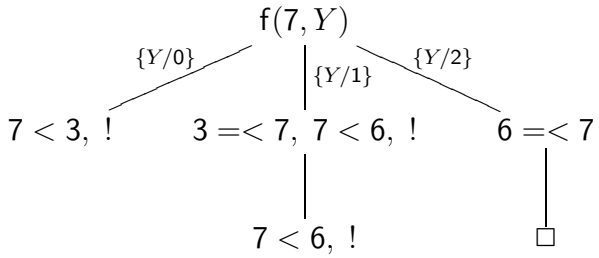
The effect of the cut in the first `f` clause is that when processing a query `f(...)`, after the resolution with the first `f` clause and the success of the proof goal `X<3`, one can no longer backtrack to attempt alternative proofs for `X<3` or for `f(...)`. Thus, a proof attempt with the second or third `f` clause is cut off. The resulting SLD tree for our query above looks like this:



Cuts as above are also called *green* cuts, because they only affect the efficiency, but not the results or the termination of the program. If one omits the cuts, one still obtains the same solutions.

The cuts in the above program have the effect that after successfully proving the conditions in the first or second `f` clause, one never tries to apply further `f` clauses. This allows an additional improvement of the above program in order to improve efficiency further. To

this end, consider the SLD tree for the query "`?- f(7,Y).`".



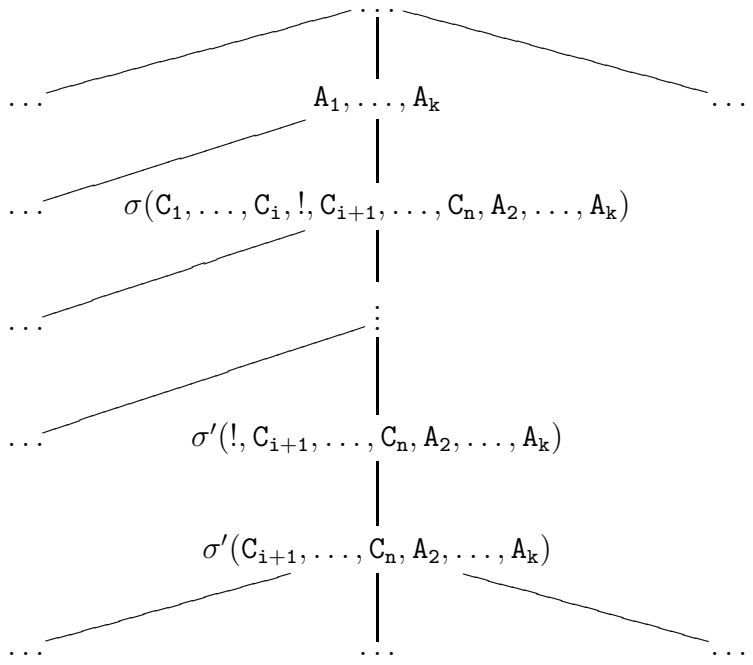
After the proof of `X<3` with the assignment of `7` to `X` has failed in the first path, in the second path one tries to prove the negated goal `3=<X` again, But this is unnecessary, because if `X<3` fails, then the proof of `3=<X` must succeed. Analogously, the proof of `X<6` fails when `X` is instantiated with `7` in the middle path. But then it is unnecessary to prove `6<=X` again in the rightmost path, as this goal must then succeed. Thus, by using cuts, the program can be changed as follows:

```
f(X,0) :- X<3, !.
f(X,1) :- X<6, !.
f(X,2).
```

These cuts are *red* cuts, because now one obtains different results when omitting them. For example, then the query "`?- f(1,Y)`" does not only have the solution `Y = 0`, but also the solutions `Y = 1` and `Y = 2`.

Also note that when adding cuts, one usually has a certain form of query in mind, where certain arguments are inputs (ground terms) and other arguments are outputs (variables). If in the above predicate, the second argument is not the output position, then undesired effects can occur. Thus the query `?- f(0,2)` results in `true`, although the desired function returns $f(0) = 0$. Similarly, the query "`?- p(X).`" for the program with the clauses "`p(0) :- !.`" and "`p(1) :- !.`" only results in the answer `X = 0`, although the solution `X = 1` would have been possible as well.

The exact meaning of the cut is as follows. If one performs resolution with a query "`?-` $\mathtt{A}_1\mathtt{,}\ldots\mathtt{,}\mathtt{A}_k$" and a program clause "`B :-` $\mathtt{C}_1\mathtt{,}\ldots\mathtt{,}\mathtt{C}_i$`, !,` $\mathtt{C}_{i+1}\mathtt{,}\ldots\mathtt{,}\mathtt{C}_n$", and afterwards the instantiated literals $\mathtt{C}_1, \ldots, \mathtt{C}_i$ are proved, then the following SLD tree is created:

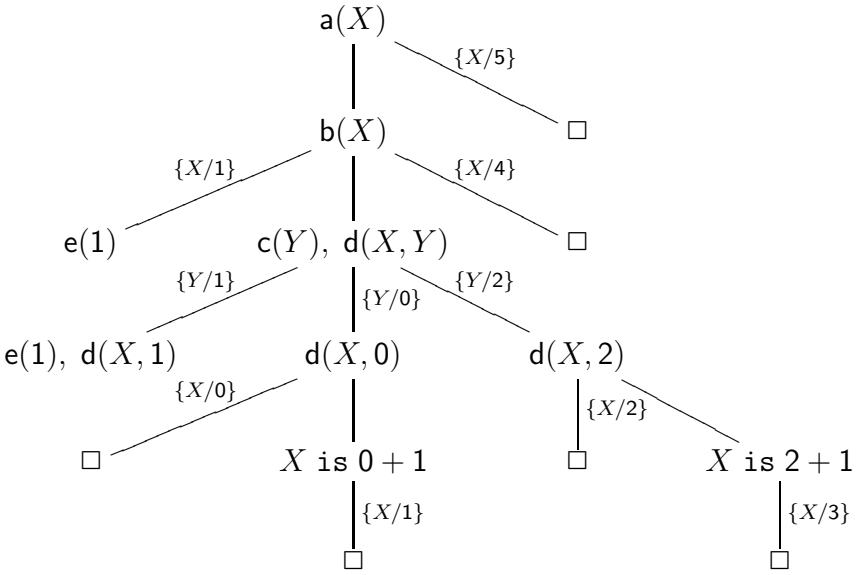


So the cut means that for all nodes between the node with the label "$\mathtt{A_1,\ldots,A_k}$" and the node with the label "$\sigma'\mathtt{(!,C_{i+1},\ldots,C_n,A_2,\ldots,A_k)}$", one does not regard any alternatives (on the right side of the tree). However, alternatives both above as well as below these nodes are considered. On the other hand, if the proof attempt fails before reaching the cut (i.e., if not all of the instantiated literals $\mathtt{C}_1$, ..., $\mathtt{C}_i$ can be proved), then one still considers alternatives as before.

This is illustrated by the following example.

```
a(X) :- b(X).
a(5).

b(1) :- e(1).
b(X) :- c(Y), d(X,Y).
b(4).

c(1) :- e(1).
c(0).
c(2).

d(X,X).
d(X,Y) :- X is Y+1.

e(0).
```

The query "`?- a(X).`" results in the following SLD tree.

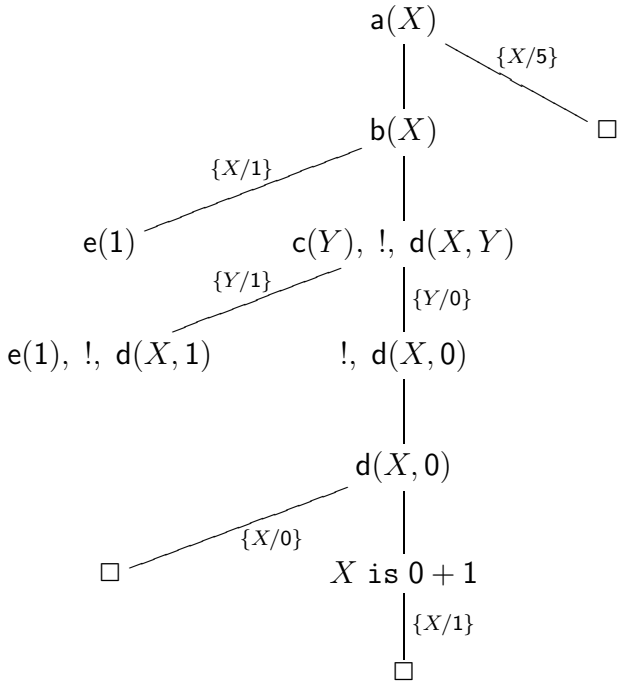


Thus, if "`;`" is entered repeatedly, the following solutions are returned: `X = 0`, `X = 1`, `X = 2`, `X = 3`, `X = 4`, `X = 5`.

Now we modify the second `b` clause by inserting a cut:

```
b(X) :- c(Y), !, d(X,Y).
```

The effect is that the alternatives for `a` and `d` are still considered, but one does not regard the alternatives for `b` and `c` (after the first success) anymore. This results in the following SLD tree:



Thus, if "`;`" is entered repeatedly, we get the following solutions: `X = 0`, `X = 1`, `X = 5`.

As an example for the use of the cut, let us consider our gcd-program from Section 5.1.

```
gcd(X,0,X).
gcd(0,X,X).
gcd(X,Y,Z) :- X =< Y, X > 0, Y1 is Y-X, gcd(X,Y1,Z).
gcd(X,Y,Z) :- Y < X, Y > 0, X1 is X-Y, gcd(X1,Y,Z).
```

If one of the first two clauses is unified with the query, one should not use any further attempts with other `gcd` clauses. Therefore we now insert a cut. This ensures that the last two clauses can only be reached if `X` and `Y` are both greater than 0 (if we restrict ourselves to natural numbers `X` and `Y`). Thus, one can then omit the literals `X > 0` and `Y > 0`. Finally, we introduce a cut behind the literal `X =< Y` in the penultimate clause. This ensures that we reach the last clause only if `Y < X`. So in the last clause, one can omit this literal.

```
gcd(X,0,X) :- !.
gcd(0,X,X) :- !.
gcd(X,Y,Z) :- X =< Y, !, Y1 is Y-X, gcd(X,Y1,Z).
gcd(X,Y,Z) :- X1 is X-Y, gcd(X1,Y,Z).
```

Finally, we present a natural example for the use of the cut. The predicate `remove(X,Xs, Ys)` holds if the list `Ys` is created from the list `Xs` by removing all occurrences of `X` from `Xs`.

```
remove(_,[],[]).
remove(X,[X|Xs],Ys) :- !,remove(X,Xs,Ys).
remove(X,[Y|Xs],[Y|Ys]) :- remove(X,Xs,Ys).
```

Then the query "`?- remove(1,[0,1,2,1],Ys).`" has the only solution `Ys = [0,2]`. Without the cut there would be the additional solutions `Ys = [0,2,1]`, `Ys = [0,1,2]`, and `Ys = [0,1,2,1]`.

5.4.2 Meta Variables and Negation

Prolog allows the use of *meta variables*. These are variables that represent *formulas* instead of *terms*. Now such variables can be instantiated as well during unification. We also have meta predicates, which can have *formulas* instead of only *terms* as arguments.

As a simple example consider the program

```
p(a).
a.
```

Here `a` is a 0-ary *predicate symbol*, i.e., the predicate symbol `p` has a formula instead of a term as its argument. For the query "`?- p(X), X.`" `X` is a meta variable that is instantiated with a formula. This query succeeds with the answer substitution `X = a`. An uninstantiated meta variable on its own must not be used for resolution (it always has to be instantiated before). Thus, a query like "`?- p(X), X, Y.`" leads to a program error.

Another example for the use of meta variables is the following program:

```
or(X,Y) :- X.
or(X,Y) :- Y.
```

This predicate is pre-defined in Prolog under the name ";". More precisely, ";" is an infix operator with the directive `:- op(1100,xfy,;).`[2] Hence, the query "`?- X = 4 ; X = 5.`" results in the answer substitution `X = 4` and if the user inputs ";", then this leads to the answer substitution `X = 5`.

By using the cut, one can implement meta predicates, which negate existing predicates or which combine them using Boolean connectives. For instance, the following program implements "`if A then B else C`".

```
if(A,B,C) :- A, !, B.
if(A,B,C) :- C.
```

Here, the cut is necessary to prevent further proofs when `A` is true and `B` fails. Otherwise, `if(A,B,C)` would always be provable whenever `C` is provable. In Prolog, a similar predicate is pre-defined. Instead of `if(A,B,C)`, here one can write "`A -> B;C`".

Another important auxiliary predicate is the *negation*. Up to now we can only derive positive statements (i.e., existentially quantified statements of the form $A_1 \wedge \ldots \wedge A_k$) from a logic program. Now our goal is to derive conjunctions which may contain negated atomic formulas $\neg A$. To realize negation in Prolog, the following assumptions are made:

- All true statements about the world can be derived from the program (*closed world assumption*). So if a statement A cannot be derived then it really does not hold and therefore $\neg A$ is true.
- If a statement cannot be derived from the program, then this is determined in finite time.

Thus, the negation is interpreted as *finite failure* (*negation as failure*). So in order to prove $\neg A$, one tries to prove A instead. If the SLD tree for A is finite and has no successful branch, then $\neg A$ is proven. The negation can be implemented as follows using the cut:

```
not(A) :- A, !, fail.
not(A).
```

Here `fail` is a pre-defined predicate, which always fails. The cut is again necessary, because otherwise `not(A)` would always be true. The predicate `not` is pre-defined in Prolog (and it can also be written as the prefix-operator `\+`).

An example for the use of negation is the definition of non-equality:

```
not_equal(X,Y) :- not(X=Y).
```

As expected, the query "`?- not_equal(1,2)`" yields the result `true` and the query "`?- not_equal(1,1)`" returns `false`. The query "`?-X=2, not_equal(1,X)`" return `X = 2`, but "`?- not_equal(1,X), X=2`" yields `false`. The reason is that the negation transforms the existential quantifier into a universal quantifier. The query `not_equal(1,X)` corresponds to

[2]The operator "`,`" for the conjunction has the precedence 1000, i.e., it binds stronger. Thus, the query "`?- p(X,Y).`" in the program with the clause "`p(X,Y) :- X = 1, Y = 1; X = 2, Y = 2.`" leads to the answer substitutions `X = 1, Y = 1` and `X = 2, Y = 2`.

the question whether *all* `X` are different from `1`. Of course, this is not the case. If however `X` is instantiated with `2` before, then this statement is true.

The negation in Prolog does not necessarily correspond to the intuitive meaning of the negation. One reason is that not every failure is noticed, because infinite unsuccessful SLD trees are not recognized as being unsuccessful. (The problem is that it is not possible to decide whether the empty clause can be derived by SLD resolution) To illustrate this, consider the following program:

```
even(0).
even(X) :- X1 is X-2, even(X1).
```

Indeed `even(1)` is not provable, but the proof tree is infinite. Thus neither the query "`?- even(1).`" nor the query "`?- not(even(1)).`" terminates.

In the version

```
even(0).
even(X) :- X >= 2, X1 is X-2, even(X1).
```

`even(t)` terminates for each number `t`, but since the query "`?- even(-2).`" is not provable, one can prove "`?- not(even(-2)).`". So the closed world assumption does not always hold (this is a problem of modeling the domain of interest completely).

An alternative version that works correctly on all integers is

```
even(0) :- !.
even(X) :- X > 0, !, X1 is X-1, not(even(X1)).
even(X) :- X1 is X+1, not(even(X1)).
```

5.5 Input and Output

Up to now, queries were the only way to transfer *input* to a program. An *output* of programs was only possible by the answer substitutions of the program or by the output `true` or `false`. But Prolog also has extra-logical predicates to perform side effects with input and output.

The predicate symbol `write/1` is used to write a term to the current output stream. By default, this is the user's screen. Thus, the query `write(t)` succeeds for every term `t` and as a side effect, `t` (or a variable renamed variant of `t`) is printed (e.g., on the screen). For example, for the query "`?- X is 2+3, write(X).`", `5` is printed as a side effect and the answer substitution is `X = 5`. The query "`?- write('This is a constant').`" prints the constant "`This is a constant`" as a side effect.

For the program

```
mult(X,Y) :- Result is X*Y, write(X*Y), write(' = '), write(Result).
```

we get the following result:

```
?- mult(3,4).
3*4 = 12
```

Note that the side effects of a `write` literal cannot be undone when backtracking. Thus, for the program

```
q(a).
q(b).
p :- q(X), write(X), X = b.
```

we obtain

```
?- p.
ab
```

Another pre-defined predicate is `nl/0` (for **n**ew**l**ine), which causes a new line in the current output. So we have

```
?- write(a),nl,write(b),nl,write(c).
a
b
c
```

Moreover, there are numerous additional predicates for output and output formatting.

For input, there is a pre-defined predicate `read/1`. Here, `read(t)` reads a term `s` from the input and then tries to unify `t` and `s`. If `t` and `s` are not unifiable, then the proof goal `read(t)` fails. To indicate the end of the term `s`, it must end with a ".".

As an example, consider the following program:

```
sqr(X,Y) :- Y is X*X.

sqr :- nl,
       write('Please enter a number or "stop":'),
       nl,
       read(X),
       proc(X).

proc(stop) :- !.
proc(X) :- sqr(X,Y),
           write('The square of '),
           write(X),
           write(' is '),
           write(Y),
           sqr.
```

To run the program, one poses the query `?- sqr.` A possible program run then looks like this.

```
?- sqr.

Please enter a number or "stop":
```

```
|: 3.
The square of 3 is 9
Please enter a number or "stop":
|: -4.
The square of -4 is 16
Please enter a number or "stop":
|: stop.
```

It is also possible to perform input and output with files. To this end, do this one has to set the current input or output stream to the respective files. For this, one can use the predicates `see/1` and `tell/1`. The query `see(t)` sets the input stream to the file named `t` (and `tell(t)` works analogously). The predicates `seen` and `told` close the input resp. output stream and set the current stream back to `user`.[3] One can now change the above program as follows:

```
sqr(X,Y) :- Y is X*X.


start :- nl,
          write('Please enter the name of an input file:'),
          nl,
          read(InputFile),
          write('Please enter the name of an output file:'),
          nl,
          read(OutputFile),
          see(InputFile),
          tell(OutputFile),
          sqr,
          seen,
          told.

sqr :- read(X),
       proc(X).

proc(end_of_file) :- !.
proc(X) :- sqr(X,Y),
           write('The square of '),
           write(X),
           write(' is '),
           write(Y),
           nl,
           sqr.
```

The program now reads terms from the input file. When the end of the file is reached,

[3] Moreover, there exist further similar predicates like `open` and `close`.

`read(X)` returns the instantiation `X = end_of_file`. If `input` is the file with the content

```
3.  -4.
```

then the following program run is possible:

```
?- start.

Please enter the name of an input file:
|: input.
Please enter the name of an output file:
|: output.
```

After that, the file `output` contains the following:

```
The square of 3 is 9
The square of -4 is 16
```

5.6 Meta Programming

In this section we introduce several pre-defined predicates of Prolog which can manipulate terms (Section 5.6.1) and which can even change the current program "during its own execution" (Section 5.6.2).

5.6.1 Processing Terms and Atomic Formulas

In Section 5.1 we already presented the pre-defined predicate `number/1` which can be used to check whether an object is a number. Prolog offers numerous other predicates in order to recognize different types of terms and atomic formulas:

- `var(t)` is true if `t` is a non-instantiated variable. For example, the query "`?- var(X).`" results in the answer `true`. On the other hand, the query "`?- X = 2, var(X).`" returns `false`.

- `nonvar(t)` is true if `t` is not a variable. Therefore, the query "`?- nonvar(a).`" returns `true` and "`?- X = 2, nonvar(X).`" returns the answer substitution `X = 2`. The query "`?- nonvar(X).`" yields `false`, although of course there would be "answer substitutions" for `X` where the instantiation of `X` is not a variable.

- `atomic(t)` is true if `t` is a 0-ary predicate or function symbol or if `t` is a number.[4] For example, the queries "`?- atomic(a).`", "`?- atomic(-).`", "`?- atomic(2).`", and "`?- atomic(-2).`" lead to `true` and "`?- atomic(X).`" and "`?- atomic(a(a)).`' yield `false`.

[4] If one wants to exclude numbers, one can use the predicate `atom/1` instead.

- `compound(t)` is true if `t` is a term or an atomic formula that does not only consist of a 0-ary function or predicate symbol or of a number. So the queries "`?- compound(a).`", "`?- compound(X).`", "`?- compound(2).`", and "`?- compound(-2).`" return the answer `false` and "`?- compound(1+2).`" and "`?- compound(a(a)).`" return `true`.

In addition to the above predicates for the *identification* of certain types of terms, Prolog also contains three predicates to extract components of terms and to construct new terms. The idea here is that terms or atomic formulas like `f(a,b)` can also be written as *lists*. The first element of the list is the outermost function or predicate symbol `f` and the remaining elements of the list are the arguments of this symbol. Thus, `f(a,b)` results in the list `[f,a,b]`. Here the first element of the list is a *0-ary* symbol `f/0` whereas the outermost symbol of the term `f(a,b)` is a *2-ary symbol* `f/2`.

To this end, in Prolog there is a pre-defined 2-ary predicate symbol "`=..`" (called "*univ*"), where "`t =.. l`" is true iff `l` is the list representation of the term or atomic formula `t`. Thus, for the query "`?- f(a,b) =.. L.`" one obtains the answer substitution `L = [f,a,b]` and for "`?- 1+2 =.. L.`" one obtains `L = [+,1,2]`. Analogously, the query "`?- T =.. [f,a,b].`" yields the answer `T = f(a,b)` and "`?- T =.. [f].`" results in `T = f`. (Queries like "`?- X =. Y.`", "`?- X =.. [Y,a,b].`", or "`?- X =.. [f|L].`" are not allowed, because here the functor or the length of the argument list are not defined. Thus, such queries cause an abort of the program.)

The transformation to list representation via "`=..`" is only done on the topmost level. So the query "`?- p(f(X),2,g(X,Y)) =.. L.`" results in `L = [p, f(X), 2, g(X,Y)]`.

As an example, we consider a program for enlarging geometric figures. For different types of figures, different function symbols have been used, e.g:

`square(Side)`
`rectangle(Side1, Side2)`
`triangle(Side1, Side2, Side3)`
`circle(Radius)`
⋮

Our goal is to implement a predicate `enlarge/3` where `enlarge(Fig, Factor, NewFig)` holds if `NewFig` is created by enlarging the figure `Fig` by the factor `Factor`. The naive solution would be to create a separate `enlarge` clause for each type of figures:

```
enlarge(square(Side),
        Factor,
        square(NewSide)) :- NewSide is Factor*Side.

enlarge(rectangle(Side1, Side2),
        Factor,
        rectangle(NewSide1, NewSide2)) :- NewSide1 is Factor*Side1,
                                          NewSide2 is Factor*Side2.
```

```
enlarge(triangle(Side1, Side2, Side3),
        Factor,
        triangle(NewS1, NewS2, NewS3)) :- NewS1 is Factor*Side1,
                                          NewS2 is Factor*Side2,
                                          NewS3 is Factor*Side3.

enlarge(circle(radius),
        Factor,
        circle(NewRadius)) :- NewRadius is Factor*Radius.
...
```

The disadvantage is an unnecessarily large programming effort. The clauses are very similar, since each time the parameters of the figure are simply multiplied by the factor `Factor`. Furthermore, this solution has the problem that all possible figure types must be known in advance.

By using the predicate "`=..`", we can implement this program in a much more elegant way by decomposing the terms and constructing new terms accordingly:

```
enlarge(Fig,Factor,NewFig) :- Fig =.. [Type | Param],
                              multiplylist(Param, Factor, NewParam),
                              NewFig =.. [Type | NewParam].

multiplylist([],_,[]).
multiplylist([X|L],Factor,[NewX|NewL]) :- NewX is Factor*X,
                                          multiplylist(L,Factor,NewL).
```

Here the variable `Type` stands for the outermost function symbol of `Fig`, i.e., for the type of the figure.

Besides the pre-defined predicate "`=..`", there are the predicates `functor/3` and `arg/3` to access the leading function symbol and the arguments of a term or of an atomic formula. Here `functor(t,f,n)` holds if `f` is the leading symbol of the term (or of the atomic formula) `t` and if `n` is the arity of `f`. Thus, the query "`?- functor(g(f(X),X,g), F, N).`" yields `F = g` and `N = 3`. The query "`?- functor(T,g,3).`" results in `T = g(X,Y,Z)`.

The formula `arg(n,t,a)` is true, if `a` is the `n`-th argument of the term `t`, where the numbering of arguments starts with 1. So the query "`?- arg(3, g(f(X),X,g), A).`" results in `A = g`. As another example, we consider the following query for the construction of a term using `functor` and `arg`:

```
?- functor(D,date,3),
   arg(1,D,29),
   arg(2,D,june),
   arg(3,D,1982).
```

The answer substitution is `D = date(29,june,1982)`.

As the final example, we define a predicate `ground/1`, where `ground(t)` is true if `t` does not contain any variables:

```
ground(T) :- nonvar(T),
             T =.. [Functor|ArgumentList],
             groundlist(ArgumentList).

groundlist([]).
groundlist([T|Ts]) :- ground(T), groundlist(Ts).
```

5.6.2 Processing Programs

A Prolog program can be considered as a data base. One can read and inspect the contents of the data base (i.e., the program text) during runtime by using the pre-defined predicate `clause/2`. The query "`?- clause(`t_1`,`t_2`)`" is true iff there is a program clause `B :-` $C_1, \ldots, C_k$, such that `clause(`t_1`,`t_2`)` and `clause(B, (`$C_1, \ldots, C_k$`))` unify. To illustrate this, consider the following program:

```
times(_,0,0).
times(X,Y,Z) :- Y > 0, Y1 is Y-1, times(X,Y1,Z1), Z is Z1+X.
```

The query "`?- clause(times(X,Y,Z),Body).`" returns the answer substitution `Y = 0`, `Z = 0`, and `Body = true`. Here `true/0` is a pre-defined predicate, which is always true. If we now enter ";", we obtain the second answer substitution `Body = Y > 0, Y1 is Y-1, times(X,Y1,Z1), Z is Z1+X`. (Note that "`,`" is a 2-ary predicate symbol which can also be used in infix notation).

Up to now, the program data base of facts and rules was considered to be "static". However, it is possible to change this data base (i.e., the Prolog program) *during its execution*. To this end, one can use the predicates `assertz/1` and `retract/1`.

The proof of `assertz(t)` always succeeds and as a side effect, *the clause* `t` *is added* at the end of the program. So when posing the query "`?- assertz(p(0)).`", the fact `p(0)` is added to the existing program. Now asking the query "`?- p(X).`" results in the answer substitution `X = 0`. Posing the query "`?- assertz(square(X,Y) :- times(X,X,Y)).`" extends the program by the corresponding rule. The query "`?- square(3,Y).`" therefore yields `Y = 9`. In addition to the predicate symbol `assertz`, there is also the predicate `asserta/1`, which inserts a clause at the beginning of the program.[5]

Such a modification of program clauses is only possible for predicate symbols which have been declared as *dynamical*. This is automatically the case for all predicates introduced by `assertz` like `p` and `square`. But if one also wants to extend our program by new clauses with `times/3` in the head, then `times/3` must be declared as *dynamic* in the program. To this end, there is a pre-defined predicate symbol `dynamic/1` which takes as argument the name of the predicate, a slash, and the arity of the predicate. The predicate `dynamic` is also declared as a prefix operator and thus, there is no need to write brackets around the argument. The `dynamic` declaration must be included as a directive in the program before using the respective predicate symbol. We therefore modify the above `times` program as follows:

[5]Moreover, there is also a predicate `assert/1` that behaves like `assertz/1`, but its use is deprecated.

```
:- dynamic times/3.

times(_,0,0).
times(X,Y,Z) :- Y > 0, Y1 is Y-1, times(X,Y1,Z1), Z is Z1+X.
```

Now for example, we can add the fact `times(X,1,X)` to the program during its execution. To this end, we pose the query "`?- asserta(times(X,1,X)).`'. If we now ask "`?- clause(times(X,Y,Z),Body).`", the first answer substitution is `X = Z`, `Y = 1`, and `Body = true`.

We can not only add new clauses to the program during execution, but we can also delete clauses from the program. To this end, one can use the predicate `retract/1`. The proof of `retract(t)` succeeds iff there is a program clause which unifies with `t`. In this case the first such program clause is deleted.

As an example, consider again the program above which was extended by the clause `times(X,1,X)` at the beginning. If one now asks the query "`?- retract(times(X,Y,X) :- Body).`", then the first unifying program clause "`times(X,1,X)`" is deleted. If we then enter ";", then the second program clause "`times(_,0,0)`" is deleted as well. If we enter ";" again, we can also delete the last program clause for `times`. Alternatively, the query "`?- retract(times(X,Y,Z)).`" could have been used. But this query can only delete the first two `times` clauses, because here the body of the clause must be `true`.

A reasonable use of predicates like `assertz` is to save computation results such that one can access them again more quickly later. An example is the following program, which stores a table containing all products `X * Y` for numbers `X` and `Y` between 0 and 9.

```
maketable :- L = [0,1,2,3,4,5,6,7,8,9],
             member(X,L),
             member(Y,L),
             Z is X * Y,
             assertz(times(X,Y,Z)),
             fail.
```

The query "`?- maketable.`" fails but by backtracking, 100 facts are added to the program which already store all multiplications of numbers between 0 and 9. The query "`?- times(X,Y,8).`" then leads to 4 possible answer substitutions: `X = 1` and `Y = 8`, `X = 2` and `Y = 4`, etc.

However, in general the predicates `assertz` and `retract` should be used with caution, as this can lead to a very bad programming style with completely incomprehensible programs.

Finally, we show how to use `assertz` to implement a predicate `findall/3` that does not only search for the first solution to a query and then waits for user input, but instead it searches for all solutions and returns them as a list. This predicate is pre-defined in Prolog.

More precisely, `findall(t,g,l)` is true if the following holds. One tries to solve the query `g` and computes *all* possible solutions (i.e., one builds up the SLD tree completely). For each solution, the found answer substitution σ is considered and σ`(t)` added to the list. Thus, `findall(t,g,l)` is true iff `l` is the list `[`σ_1`(t), ..., `σ_n`(t)]`, where $\sigma_1, \ldots, \sigma_n$ are the answer substitutions that are found in the SLD tree during depth-first search from left to right. As an example, we consider the following program:

```
motherOf(monika, karin).
motherOf(monika, klaus).
motherOf(renate, susanne).
motherOf(renate, peter).
motherOf(susanne, aline).
motherOf(susanne, dominique).

married(werner, monika).
married(gerd, renate).
married(klaus, susanne).

fatherOf(V,K) :- married(V,F), motherOf(F,K).
```

The query "`?- findall(K, fatherOf(gerd,K), L).`" means that `L` is instantiated with the list of *all* children of `gerd`. The answer substitution is `L = [susanne, peter]`. The query "`?- findall(fatherOf(gerd,K), fatherOf(gerd,K), L).`" instead results in `L = [fatherOf(gerd, susanne), fatherOf(gerd, peter)]`.

To implement `findall` ourselves, one can use the following program:

```
findall(X, Query, Xlist) :- Query,
                            assertz(solution(X)),
                            fail
                            ;
                            collectSolutions(Xlist).

collectSolutions([X | Rest]) :- retract(solution(X)),
                                !,
                                collectSolutions(Rest).
collectSolutions([]).
```

In the clause for `findall`, one first solves the query and the first solution found is written to the data base (as the argument of the 1-ary predicate symbol `solution`). Afterwards, the overall query fails and thus, backtracking is enforced. In this way, the entire SLD tree for the query is constructed and all solutions are stored. Finally, `collectSolutions` is called. One tries to delete solutions from the data base and to add them to the result list as long as possible. The cut after `retract(solution(X))` is necessary to avoid backtracking and computing a new solution when searching for further solutions.

Since Prolog programs themselves can be regarded as terms again (by using the predicate symbol `clause`), Prolog is also suitable for writing *meta programs*. These are programs like compilers or interpreters, which process other programs. In particular, one can also implement *meta interpreters*, i.e., interpreters for a programming language which are themselves written in this programming language. So one can write a Prolog interpreter in Prolog. Prolog is also particularly suitable for *rapid prototyping*, e.g. to write prototypical interpreters for new programming languages or for new evaluation strategies in programming languages.

We now consider several meta interpreters for Prolog which have extended functionality compared to the usual Prolog interpreter. The simplest meta interpreter can be implemented as follows:

```
prove(Goal) :- Goal.
```

However, this meta interpreter is quite useless, because it delegates the entire "work" to the original Prolog interpreter.

An alternative meta interpreter is the following. It is only suitable for pure logic programs (without pre-defined predicates). We will take it as a starting point to develop modified meta interpreters.

```
prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal1), prove(Goal2).
prove(Goal) :- clause(Goal, Body), prove(Body).
```

Here, the cuts are necessary to avoid undesired effects when backtracking. Without the cuts, for the program

```
p(0).
```

and the query "`?- prove(p(X)).`", we would obtain a program error after the first solution `X = 0`, because now the third clause is used and thus, the goal `clause(true, Body)` has to be solved. But `clause` must not be called for the pre-defined symbol `true`. If only the first cut were used, then the query "`?- prove((p(X), p(X))).`" would lead to infinitely many solutions `X = 0`, since the comma is defined by the clause "`X, Y :- X, Y`".

For example, now we can implement a variant of the meta interpreter which processes literals in queries not from left to right, but from right to left:

```
prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal2), prove(Goal1).
prove(Goal) :- clause(Goal, Body), prove(Body).
```

Another variant would be a meta interpreter which returns the length of the proof:

```
prove(true,0) :- !.
prove((Goal1,Goal2),N) :- !, prove(Goal1,N1), prove(Goal2,N2), N is N1+N2.
prove(Goal,N) :- clause(Goal,Body), prove(Body,N1), N is N1+1.
```

Numerous other variations are possible. An example would be a meta interpreter which enforces termination by only performing proofs which are shorter than a given length. Another possibility would be a meta interpreter which returns the proof tree, etc.

5.7 Difference Lists and Definite Clause Grammars

Prolog provides its own notation for rules of context-free grammars, so-called *definite clause grammars*. To implement them as efficiently as possible, a special representation for lists (so-called *difference lists*) is used. We first introduce difference lists in Section 5.7.1 and then present definite clause grammars in Section 5.7.2.

5.7.1 Difference Lists

Many operations on lists can be implemented much more efficiently using a different list representation (so-called *difference lists*). To illustrate this, we again consider the previous implementation of the predicate `app/3` for list concatenation of Section 5.2.

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

The runtime complexity for the concatenation of two lists with this program is $O(n)$, where n is the length of the first list. For example, the proof of the query "`?- app([1,2,3], [4,5], Zs).`" needs four resolution steps to compute the answer substitution `Zs = [1,2,3,4,5]`. The reason is that the first list is traversed until the empty list is reached.

Our goal now is an alternative list representation where list concatenation can be done in only one computation step. To this end, we now represent lists as *difference lists*, i.e., as the difference between two lists. For example, the list `[1,2,3]` can be represented by the two lists `[1,2,3,4,5]` and `[4,5]` because

```
[1,2,3,4,5] - [4,5] = [1,2,3]
```

holds. With this representation, the second list (i.e., `[4,5]`) must of course be a suffix of the first list (i.e., `[1,2,3,4,5]`). Clearly, this representation is not unique. The list `[1,2,3]` can also be represented as the difference of `[1,2,3]` and `[]` or as the difference of `[1,2,3,4,5 | Ys]` and `[4,5 | Ys]` or as the difference of `[1,2,3 | Ys]` and `Ys`. This last representation is the most general representation of difference lists.

Now we can define the following alternative implementation of `app`, which consists of only one single fact:

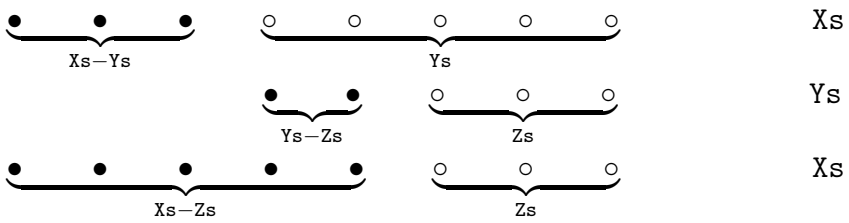
```
app(Xs - Ys, Ys, Xs).
```

Now consider a query "`?- app(`t_1`, `t_2`, Xs)`". If the first argument t_1 of `app` is a list in the most general difference list representation like "`[1,2,3 | Ys] - Ys`", then `Ys` is instantiated in a single resolution step with the second `app` argument t_2 and the result is the answer substitution `Xs = [1,2,3 |` t_2`]`. Thus, the query "`?- app([1,2,3 | Ys] - Ys, [4,5], Xs)`" leads to the answer substitution `Ys = [4,5]` and `Xs = [1,2,3,4,5]` after only one single resolution step. So now the runtime complexity is no longer linear, but constant (i.e., $O(1)$).

But a disadvantage of the above algorithm is that only the first argument of `app` is represented as a difference list. Thus, the result of `app` cannot be directly used as input for `app` in the first argument again. Therefore we now formulate a version of `app` where all three arguments are difference lists.

```
app(Xs - Ys, Ys - Zs, Xs - Zs).
```

The following diagram illustrates how the above program works:



If the most general representations of difference lists are used, then this program again computes the list concatenation in constant time and returns the resulting list also in the most general difference list representation. Thus, the query "`?- app([1,2,3 | Ys] - Ys, [4,5 | Zs] - Zs, Res).`" results in the answer substitution `Ys = [4,5 | Zs]`, `Res = [1,2,3,4,5 | Zs] - Zs`.

Note that the above program can only concatenate the lists if one chooses a difference list representation where the first two `app` arguments are "compatible". A query "`?- app(`t_1 `-` t_2`,` s_1 `-` s_2`, Res).`" fails if t_2 and s_1 are not unifiable. So the query "`?- app([1,2,3,6] - [6], [4,5] - [], Res).`" gives the answer `false`. Similarly, the following query results in `false`:

```
?- L = [1|Ys] - Ys,
   app(L, [2 | Zs] - Zs, Res1),
   app(L, [3 | Ws] - Ws, Res2).
```

The reason is that after solving the second atom in the query, the variable `L` is instantiated with `[1,2 | Zs] - [2 | Zs]` and `Ys` is instantiated with `[2 | Zs]`. But the terms `[2 | Zs]` and `[3 | Ws]` are not unifiable.

As another example we consider an algorithm `flatten/2` for the conversion of nested lists to linear lists. (This predicate is pre-defined in Prolog.) Here, `append/3` is the normal (and pre-defined) predicate for concatenation of lists in the classical representation (i.e., in the representation without using difference lists).

```
flatten([X|Xs],Ys) :- !,
                      flatten(X, X1),
                      flatten(Xs,Xs1),
                      append(X1,Xs1,Ys).
flatten([],[]) :- !.
flatten(X,[X]).
```

For example, the query "`?- flatten([[1,2],[[3]]] [[4,[5,6]]]],Ys).`" returns the answer `Ys = [1,2,3,4,5,6]`. The runtime complexity to compute "`?- flatten(t,Ys)`" for a list `t` with n elements is $O(n^2)$, because each call of `append` has linear runtime complexity.

To reduce the complexity, one should therefore use the version of `append` based on difference lists.

```
flatten([X|Xs],Ys - Zs) :- !,
                           flatten(X, X1 - X2),
```

```
                          flatten(Xs, Xs1 - Xs2),
                          app(X1 - X2, Xs1 - Xs2, Ys - Zs).
flatten([],Ys - Ys) :- !.
flatten(X,[X|Ys] - Ys).
```

The query "`?- flatten([[1,2],[[3]] ] [[4,[5,6]]]],Ys).`" now yields the answer `Ys = [1,2,3,4,5,6|Ws] - Ws`. Since `app` queries can now be solved with constant runtime complexity, the total runtime complexity is only $O(n)$. If one does not want to obtain a result in difference list representation, one can use the additional clause

```
flat(Xs,Ys) :- flatten(Xs, Ys - []).
```

The query "`flat([[1,2],[[3]]] [[4,[5,6]]]],Ys).`" results in the answer `Ys = [1,2,3,4,5,6]`.

Note that the above `flatten` program can still be simplified. The atom `app(X1 - X2, Xs1 - Xs2, Ys - Zs)` in the body of the first clause succeeds iff `X2` and `Xs1` are unifiable and both `X1` and `Ys` as well as `Xs2` and `Zs` are unifiable. We can therefore directly replace `X2` by `Xs1`, `X1` by `Ys`, and `Xs2` by `Zs`. This results in the following program:

```
flatten([X|Xs],Ys - Zs) :- !,
                          flatten(X, Ys - Xs1),
                          flatten(Xs, Xs1 - Zs).
flatten([],Ys - Ys) :- !.
flatten(X, [X|Ys] - Ys).
```

5.7.2 Definite Clause Grammars

We now show how to represent context-free grammars in Prolog such that one directly obtains an algorithm that solves the word problem (i.e., an algorithm that decides for each word whether it is in the language or not). In this way, parsers for different (programming) languages can easily be implemented in Prolog.

A context free grammar G is a 4-tuple $G = (N, T, S, P)$ where N is a set of non-terminal symbols and T is a set of terminal symbols. $S \in N$ is the start symbol and P is a set of productions (or "rules") of the form $A \to \alpha$ with $A \in N$ and $\alpha \in (N \cup T)^*$. The corresponding derivation relation $\Rightarrow_G$ between words from $(N \cup T)^*$ is defined as $\beta \Rightarrow_G \gamma$ if there is a production $A \to \alpha$ in P such that $\beta = \beta_1\ A\ \beta_2$ and $\gamma = \beta_1\ \alpha\ \beta_2$. The *language* of G is $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$.

As an example, we consider the following grammar $G = (N, T, S, P)$:

- $N =$ { Sentence, Nominalphrase, Verbalphrase, Article, nNun, Verb }
- $T =$ { a, the, cat, mouse, scares, hates }
- $S =$ Sentence

- P consists of the following rules:

$$\begin{aligned}
\text{Sentence} &\to \text{Nominalphrase Verbalphrase}\\
\text{Nominalphrase} &\to \text{Article Noun}\\
\text{Verbalphrase} &\to \text{Verb}\\
\text{Verbalphrase} &\to \text{Verb Nominalphrase}\\
\text{Article} &\to \text{a}\\
\text{Article} &\to \text{the}\\
\text{Noun} &\to \text{cat}\\
\text{Noun} &\to \text{mouse}\\
\text{Verb} &\to \text{scares}\\
\text{Verb} &\to \text{hates}
\end{aligned}$$

For example, the language of this grammar contains the following words:

```
a cat scares the mouse
the mouse hates the cat
a mouse scares a mouse
a mouse hates
```

The following conventions are used for the representation in Prolog:

- non-terminal symbols of N are represented by constants (i.e., by 0-ary predicate symbols).
- terminal symbols of T are written as singleton lists with a constant.
- words from T^* are written as lists of constants. The empty word ε is written as `[]`.
- Words from $(N \cup T)^*$ are written as comma-separated sequences of constants and lists of constants.
- In rules, one writes `-->` instead of $\to$.

Thus, the above grammar can be written as a Prolog program as follows:

```
sentence --> nominalphrase, verbalphrase.
nominalphrase --> article, noun.
verbalphrase --> verb.
verbalphrase --> verb, nominalphrase.
article --> [a].
article --> [the].
noun --> [cat].
noun --> [mouse].
verb --> [scares].
verb --> [hates].
```

The word "a mouse Verbalphrase" would be written as "`[a, mouse], verbalphrase`'.

Prolog has to translate the above representation into normal clauses. The goal is to turn it into a Prolog program which we can check for each word whether it belongs to the respective language.

A first idea would be to assign a 1-ary predicate to each non-terminal symbol, which tests whether its argument can be derived from the corresponding non-terminal symbol using the rules of the grammar.

Then, a grammar rule of the form "`a --> [a1, a2, a3]`" (with non-terminal symbol `a` and terminal symbols `a1`, `a2`, `a3`) is converted to the fact

```
a([a1, a2, a3]).
```

For example, in the example above one obtains the fact `verb([scares])`. Analogously, a rule of the form "`a --> a1`" is translated to

```
a(A) :- a1(A).
```

and "`a --> a1, a2`" is translated into the following rule.

```
a(A) :- append(A1, A2, A),
        a1(A1),
        a2(A2).
```

So in our example one obtains the rule

```
sentence(S) :- append(NP, VP, S), nominalphrase(NP), verbalphrase(VP).
```

However, here `append` is always executed with non-instantiated first arguments which leads to a very inefficient program.

Thus, it would be better to represent lists as difference lists again. Then `a(A - B)` would hold if one can derive the list of terminal symbols `A` without its suffix `B` from the non-terminal symbol `a`. For the automatic translation of grammar rules into clauses, Prolog uses a notation where difference lists are not represented in the form "`A - B`", but *two* arguments are used instead. So now we use a *2-ary* predicate symbol for each non-terminal symbol `a`, where `a(A, B)` holds if the list of terminal symbols `A` without its suffix `B` can be derived from the non-terminal symbol `a`. A rule of the form "`a --> a1`" is now converted to

```
a(A,B) :- a1(A,B)
```

A first approach to translate a rule "`a --> a1, a2`" would yield the following rule. Here, `app/3` is the implementation of list concatenation with difference lists.

```
a(A,B) :- app(Xs - Ys, Vs - Ws, A - B),
          a1(Xs, Ys),
          a2(Vs, Ws).
```

The `app` atom is true iff `Xs` and `A`, `Ws` and `B`, and `Ys` and `Vs` are unifiable. The above rule can therefore be reformulated as

```
a(A,B) :- a1(A, C),
          a2(C, B).
```

A grammar rule of the form "`a --> [a1, a2, a3]`" (with non-terminal symbol `a` and terminal symbols `a1`, `a2`, `a3`) is translated into the following fact:

```
a([a1, a2, a3 | Xs], Xs).
```

A similar translation is also possible if one or more terminal symbols are in front of a non-terminal symbol on the right-hand side. A rule of the form "`a --> a1, [a2, a3], a4`" can be translated into the following clause:

```
a(A,B) :- a1(A, [a2, a3 | C]),
          a4(C, B).
```

Thus, the above Prolog program with the example grammar would be translated into the following clauses:

```
sentence(S, R) :- nominalphrase(S, VP), verbalphrase(VP, R)
nominalphrase(NP, R) :- article(NP, N), noun(N, R).
verbalphrase(VP, R) :- verb(VP, R).
verbalphrase(VP, R) :- verb(VP, NP), nominalphrase(NP, R).
article([a | R], R).
article([the | R], R).
noun([cat | R], R).
noun([mouse | R], R).
verb([scares | R], R).
verb([hates | R], R).
```

The query "`?- sentence([the, cat, scares, a, mouse],[]).`" now returns `true`, as expected. Analogously, we also get the answer `true` for the query "`?- sentence([the, cat, scares, a, mouse, trash],[trash]).`".

One can also let Prolog compute all elements of the language. The query "`?- sentence(S, []).`" results in

```
S = [a, cat, scares] ;
S = [a, cat, hates] ;
S = [a, cat, scares, a, cat] ;
S = [a, cat, scares, a, mouse] ;
...
```

The notation of the definite clause grammars also offers extensions to describe context dependencies or to construct syntax trees that are needed when parsing programming languages.

Chapter 6

Constraint Logic Programming

After having considered both "pure" logic programming and its implementation in the language Prolog, we want to conclude with an important extension of logic programming, so-called "*constraint logic programming*" (CLP). In Section 6.1, we first introduce logic programming with constraints formally. Then in Section 6.2, we discuss how to solve the integration of constraints in the programming language Prolog. For further literature on this topic we refer to [FA10, MS98], for example.

6.1 Syntax and Semantics of Constraint Logic Programs

A *constraint* is a restriction or condition. Such conditions are typically formulated as atomic formulas. The following definition formally introduces the notion of constraints. Our goal is to extend logic programs over a signature (Σ, Δ) by constraints over of a sub-signature (Σ', Δ'). Here, it is advantageous to treat the equality predicate symbol "=" and the special predicate symbols true and fail separately.

Definition 6.1.1 (Constraint Signature and Constraints) *Let (Σ, Δ) be a signature with $\mathsf{true}, \mathsf{fail} \in \Delta_0$ and $= \in \delta_2$. Let $\Sigma' \subseteq \Sigma$ and $\Delta' \subseteq \Delta$ be subsets of the signature, where Δ' contains none of the predicate symbols true, fail, or $=$. Then $(\Sigma, \Delta, \Sigma', \Delta')$ is a* constraint signature*. The atomic formulas from $\mathcal{A}t(\Sigma', \Delta', \mathcal{V}) \cup \mathcal{A}t(\Sigma, \{=\}, \mathcal{V}) \cup \{\mathsf{true}, \mathsf{fail}\}$ are called* constraints *over the signature $(\Sigma, \Delta, \Sigma', \Delta')$.*

Thus, a constraint signature determines a subset Σ' of the function symbols and a subset Δ' of the predicate symbols that are used to construct constraints. Predicates from Δ' can only be applied to terms which just contain function symbols from Σ'. The equality predicate $=$ can be applied to any terms. All atomic formulas with predicates from $\Delta' \cup \{\mathsf{true}, \mathsf{fail}, =\}$ are then called *constraints*.

Example 6.1.2 *As an example, we consider a constraint signature $(\Sigma, \Delta, \Sigma', \Delta')$ where Σ' and Δ' contain function and predicate symbols for operating on integers:*

$$\begin{array}{lcl} \Sigma'_0 & = & \mathbb{Z} \\ \Sigma'_1 & = & \{-, \mathsf{abs}\} \\ \Sigma'_2 & = & \{+, -, *, /, \mathsf{mod}, \mathsf{min}, \mathsf{max}\} \\ \Delta'_2 & = & \{\#>=,\ \#=<,\ \#=,\ \#\backslash=,\ \#>,\ \#<\} \end{array}$$

The sets Σ' and Δ' are also called Σ_{FD} and Δ_{FD}, where "FD" stands for Finite Domains. If $\mathsf{f} \in \Sigma_1$, then we get the following constraints, among others:

$$\begin{array}{rcl} X + Y & \#> & Z * 3 \\ \mathsf{max}(X, Y) & \#= & X \bmod 2 \\ \mathsf{f}(X) + 2 & = & Y + Z \end{array}$$

To decide when a constraint is true, one also needs a *constraint theory* CT. Here, CT is a set of formulas and a constraint is true iff it is entailed by the set CT.

Definition 6.1.3 (Constraint Theory) *Let $(\Sigma, \Delta, \Sigma', \Delta')$ be a constraint signature. If $CT \subseteq \mathcal{F}(\Sigma', \Delta', \mathcal{V})$ is satisfiable and only contains closed formulas, then CT is called a* constraint theory.

Example 6.1.4 *Let $S_{FD} = (\mathbb{Z}, \alpha)$ be the structure with carrier $\mathbb{Z}$ and the "intuitive" meaning α of all function and predicate symbols. So we have $\alpha_n = n$ for all $n \in \mathbb{Z}$. Furthermore, α_+ is the addition function, etc., and $\alpha_{\#>}$ is the set of all pairs of numbers (n, m) with $n > m$, etc. The intuitive constraint theory for the signature of Example 6.1.2 is the set CT_{FD} of all closed formulas $\varphi \in \mathcal{F}(\Sigma_{FD}, \Delta_{FD}, \mathcal{V})$ where:*

$$\varphi \in CT_{FD} \quad \textit{iff} \quad S_{FD} \models \varphi$$

Note that CT_{FD} is not only infinite, but not decidable (and not even semi-decidable). The reason is that we have also included functions like multiplication in Σ_{FD}. If we restricted ourselves to addition only (this is called Presburger arithmetic*), then we could define CT_{FD} as a finite set. In other words, then there exists a finite set of axioms which entail exactly all true formulas over the integers.*

We now define the syntax and semantics of logic programs with constraints. The syntax does not differ from the syntax of normal logic programs. Here, we assume that logic programs always already contain the clauses to define true and $=$, and that there is no clause to define fail. The predicate symbols of the constraints must not appear on the left-hand sides of other rules. Furthermore, the predicate symbols from Δ' may only be applied to terms whose function symbols are from Σ'.

Definition 6.1.5 (Syntax of Constraint Logic Programs) *A non-empty finite set $\mathcal{P}$ of definite Horn clauses over a constraint signature $(\Sigma, \Delta, \Sigma', \Delta')$ is a* logic program with constraints *over $(\Sigma, \Delta, \Sigma', \Delta')$ if $\{\mathsf{true}\} \in \mathcal{P}$ and $\{X = X\} \in \mathcal{P}$ and if for all other clauses $\{B, \neg C_1, \ldots, \neg C_n\} \in \mathcal{P}$ we have:*

(a) If $B = p(t_1, \ldots, t_m)$*, then* $p \notin \Delta' \cup \{\mathsf{true}, \mathsf{fail}, =\}$.

(b) If $C_i = p(t_1, \ldots, t_m)$ *and* $p \in \Delta'$*, then* $t_j \in \mathcal{T}(\Sigma', \mathcal{V})$ *for all* $1 \leq j \leq m$.

Moreover, Condition (b) must also hold for all queries $\{\neg C_1, \ldots, \neg C_n\}$.

Example 6.1.6 *We again consider a constraint signature* $(\Sigma, \Delta, \Sigma_{FD}, \Delta_{FD})$*. An example for a logic program with constraints is the following set of clauses* $\mathcal{P}$*. Here we use again the common notation for logic programs (as facts and rules instead of clauses).*

```
fact(0,1).
fact(X,Y) :- X #> 0, X1 #= X-1, fact(X1,Y1), Y #= X*Y1.
```

One can see the similarity to the program for computing the factorial in Section 5.1:

```
fac(0,1).
fac(X,Y) :- X > 0, X1 is X-1, fac(X1,Y1), Y is X*Y1.
```

Now we define the semantics of logic programming with constraints. As in Section 4.1.1 and 4.1.2 we will do this both in a declarative as well as an operational way. (A fixpoint semantics of constraint logic programming would be possible, too).

The declarative semantics of constraint logic programming is very intuitive. In ordinary logic programming, all instantiations of a query which are entailed by the clauses of the program $\mathcal{P}$ are considered to be "true". So here, the program clauses are treated as *axioms*. In constraint logic programs, the axioms from $\mathcal{P}$ are simply extended by the additional axioms CT.

Definition 6.1.7 (Declarative Semantics of Constraint Logic Programs) *Let* $\mathcal{P}$ *be a constraint logic program and let* CT *be a corresponding constraint theory. Let* $G = \{\neg A_1, \ldots, \neg A_k\}$ *be a query. Then the* declarative semantics *of* $\mathcal{P}$ *and* CT *w.r.t.* G *is*

$$D[\![\mathcal{P}, CT, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid \mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_k),\ \sigma \text{ is a ground substitution}\}.$$

Example 6.1.8 *Consider the program* $\mathcal{P}$ *from Example 6.1.6, the constraint theory* CT_{FD} *from Example 6.1.4, and the query* $G = \{\neg\mathsf{fact}(1, Z)\}$*. The only ground substitution with* $\mathcal{P} \cup CT_{FD} \models \sigma(\mathsf{fact}(1, Z))$ *is* $\sigma(Z) = 1$*. Thus,* $D[\![\mathcal{P}, CT, G]\!] = \{\mathsf{fact}(1, 1)\}$*. For the query* $G' = \{\neg\mathsf{fact}(X, 1)\}$*, we have* $\mathcal{P} \cup CT_{FD} \models \sigma_1(\mathsf{fact}(X, 1))$ *and* $\mathcal{P} \cup CT_{FD} \models \sigma_2(\mathsf{fact}(X, 1))$ *for the substitutions* $\sigma_1(X) = 0$ *and* $\sigma_2(X) = 1$*. Therefore*

```
?- fact(1,Z).
```

results in `Z = 1` *and the query*

```
?- fact(X,1).
```

results in `X = 0` *and* `X = 1`.

Def. 6.1.7 clearly shows that ordinary logic programming is a special case of constraint logic programming. Obviously, logic programs with empty constraint theory CT correspond to the ordinary logic programs considered so far.

Corollary 6.1.9 *Let $\mathcal{P}$ be a constraint logic program with $\Sigma' = \varnothing$ and $\Delta' = \varnothing$. Then for all queries G, we have $D[\![\mathcal{P}, \varnothing, G]\!] = D[\![\mathcal{P}, G]\!]$.*

To explain how constraint logic programs are *evaluated*, we now define their procedural semantics. Unfortunately, this is a bit more involved than the declarative semantics. The problem is that the axioms in CT can be arbitrary formulas (and not necessarily only definite Horn clauses). Whether a query is entailed by the program clauses of $\mathcal{P}$ can be examined with binary SLD resolution. However, this is not possible for the axioms in CT. Instead, we assume that we have a technique that checks entailment from CT automatically. The problem now is how to combine this technique (for checking entailment from CT) with SLD resolution (for checking entailment from $\mathcal{P}$). Here, the idea is to also represent the SLD resolution steps with the help of constraints. In this way, we obtain a unified representation of both types of proof steps (the steps that perform SLD resolution with clauses from $\mathcal{P}$ and the steps that solve constraints using the constraint theory CT). We will therefore explicitly use equations as constraints in order to represent information from the applied unifiers. As the following example shows, one could use this representation also for ordinary logic programs without constraints.

Example 6.1.10 *We consider the following (pure) logic program from Section 5.1.*

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

In the previous definition of the procedural semantics, a configuration (G, σ) consisted of the current query G and the already computed substitution. The computation started with the empty (or "identical") substitution $\varnothing$. When posing the query $\{\neg\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U)\}$, we resulted in

$$\begin{array}{rl} & (\neg\mathsf{add}(\mathsf{s}(0), \mathsf{s}(0), U),\ \varnothing) \\ \vdash_{\mathcal{P}} & (\neg\mathsf{add}(\mathsf{s}(0), 0, Z),\ \{X/\mathsf{s}(0),\ Y/0,\ U/\mathsf{s}(Z)\}) \\ \vdash_{\mathcal{P}} & (\square,\ \underbrace{\{X'/\mathsf{s}(0),\ Z/\mathsf{s}(0)\} \circ \{X/\mathsf{s}(0),\ Y/0,\ U/\mathsf{s}(Z)\}}_{\{X'/\mathsf{s}(0),\ Z/\mathsf{s}(0),\ X/\mathsf{s}(0),\ Y/0,\ U/\mathsf{s}(\mathsf{s}(0))\}}) \end{array}$$

From the substitution obtained at the end, the answer substitution is obtained by restricting it to the variable U from the original query. The result is the answer substitution $\{U/\mathsf{s}(\mathsf{s}(0))\}$.

Now the idea is not to perform unification directly, but to just collect the unification constraints of the corresponding unifiers instead. Here "$\overline{A = B}$" represents the constraint that the two atomic formulas A and B have to be unified. The configurations now are of the form (G, CO), where CO is a conjunction of unification constraints of the form "$\overline{A = B}$". Here one starts with the empty conjunction, i.e., with the formula true*, which is always true.*

$$\begin{array}{ll} & (\neg \mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U),\ \mathsf{true}) \\ \vdash_{\mathcal{P}} & (\neg \mathsf{add}(X,Y,Z),\ \overline{\mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U) = \mathsf{add}(X,\mathsf{s}(Y),\mathsf{s}(Z))}) \\ \vdash_{\mathcal{P}} & (\Box,\ \overline{\mathsf{add}(X,Y,Z) = \mathsf{add}(X',0,X')} \wedge \overline{\mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U) = \mathsf{add}(X,\mathsf{s}(Y),\mathsf{s}(Z))}) \end{array}$$

The conjunction of unification problems obtained in the end can of course be further simplified. It is equivalent to the condition

$$X' = \mathsf{s}(0) \ \wedge\ Z = \mathsf{s}(0) \ \wedge\ X = \mathsf{s}(0) \ \wedge\ Y = 0 \ \wedge\ U = \mathsf{s}(\mathsf{s}(0)) \tag{6.1}$$

This corresponds to the substitution obtained in the procedural semantics of logic programs up to now.

The difference to the procedural semantics of ordinary logic programs from Definition 4.1.5 is that now we do not consider configurations of the form (G, σ) anymore, where σ is a substitution. Instead, now the configurations are of the form (G, CO). Here, CO should be a conjunction of constraints, i.e., CO may contain equations between terms. However, the equality predicate symbol cannot be applied to two atomic formulas (i.e., one cannot write $A = B$). For this reason, we define $\overline{A = B}$ as an abbreviation for a corresponding conjunction of equations between terms.

Definition 6.1.11 (Equality of Atoms) *Let A and B be atomic formulas. Then we define the formula $\overline{A = B}$ as follows:*

- *$\overline{A = B}$ is the formula* fail*, if $A = p(\ldots)$, $B = q(\ldots)$ with $p \neq q$*
- *$\overline{A = B}$ is the formula* true*, if $A = B = p$*
- *$\overline{A = B}$ is the formula $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$, if $A = p(s_1, \ldots, s_n)$ and $B = p(t_1, \ldots, t_n)$*

Example 6.1.12 *With the above definition of $\overline{A = B}$, the computation steps of Example 6.1.10 now have the following form:*

$$\begin{array}{ll} & (\neg \mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U),\ \mathsf{true}) \\ \vdash_{\mathcal{P}} & (\neg \mathsf{add}(X,Y,Z),\ \underbrace{\overline{\mathsf{add}(\mathsf{s}(0),\mathsf{s}(0),U) = \mathsf{add}(X,\mathsf{s}(Y),\mathsf{s}(Z))}}_{\mathsf{s}(0)=X \,\wedge\, \mathsf{s}(0)=\mathsf{s}(Y) \,\wedge\, U=\mathsf{s}(Z)}) \\ \vdash_{\mathcal{P}} & (\Box,\ \underbrace{\overline{\mathsf{add}(X,Y,Z) = \mathsf{add}(X',0,X')}}_{X=X' \,\wedge\, Y=0 \,\wedge\, Z=X'} \wedge\ \mathsf{s}(0) = X \ \wedge\ \mathsf{s}(0) = \mathsf{s}(Y) \ \wedge\ U = \mathsf{s}(Z)) \end{array}$$

In the end, we obtain the following conjunction of constraints:

$$X = X' \ \wedge\ Y = 0 \ \wedge\ Z = X' \ \wedge\ \mathsf{s}(0) = X \ \wedge\ \mathsf{s}(0) = \mathsf{s}(Y) \ \wedge\ U = \mathsf{s}(Z) \tag{6.2}$$

To simplify the resulting conjunctions of constraints CO, a function "simplify" can be used, which converts conjunctions of constraints into equivalent conjunctions of (simpler) constraints. Of course, here one has to clarify what we mean by "equivalence". The only

possible predicate symbols in CO so far are true, fail and $=$. Therefore, when simplifying CO, one should consider the axioms over true, fail, and $=$. For the "simplify" function, we therefore require that

$$\{\forall X\ X = X, \mathsf{true}\} \models \forall\,(CO \leftrightarrow \text{simplify}(CO)).$$

For any quantifier-free formula φ with the variables $X_1, \ldots, X_n$, $\forall\,\varphi$ denotes the *universal closure* of φ, i.e., the formula $\forall X_1, \ldots, X_n\,\varphi$. Similarly, $\exists\,\varphi$ is the *existential closure* of φ, i.e., the formula $\exists X_1, \ldots, X_n\,\varphi$.

In Example 6.1.12, we could have simplify((6.2)) = (6.1), as we obviously already have

$$\forall X\ X = X \quad \models \quad \forall X', X, Y, Z, U\ \ (6.2) \leftrightarrow (6.1).$$

Of course, one can also use "simplify" after each computation step in order to simplify the current conjunctions of constraints.

Note that instead of the unifiers now we only collect the equations between the terms that have to be unified. This indeed leads to an equivalent approach, because the axiom "$\forall X\ X = X$" ensures that two terms are only considered "equal" if they are syntactically equal. Thus, the formula "$\forall X\ X = X$" serves as the "axiomatization of unification". This is expressed by the following lemma, which we will need to prove the equivalence of the declarative and the procedural semantics.

Lemma 6.1.13 (Equality and Unification) *For all terms s, t, all atoms A, B, and all substitutions σ we have:*

(a) $\forall X\ X = X \models \sigma(s = t)$ *iff* $\sigma(s)$ *and* $\sigma(t)$ *are syntactically identical.*

(b) $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A = B})$ *iff* $\sigma(A)$ *and* $\sigma(B)$ *are syntactically identical.*

Proof.

(a) We first show the direction from right to left. Let I be an interpretation that is a model of $\forall X\ X = X$. Since $\sigma(s) = \sigma(t)$ and thus $I(\sigma(s)) = I(\sigma(t))$, we also have $I \models \sigma(s) = \sigma(t)$.

Now we prove the direction from left to right. We consider the interpretation $I = (\mathcal{T}(\Sigma, \mathcal{V}), \alpha, \beta)$ with $\alpha_f = f$ for all $f \in \Sigma$, $\alpha_= = \{(r, r) \mid r \in \mathcal{T}(\Sigma, \mathcal{V})\}$, and $\beta(X) = X$. Here, $I(r) = r$ holds for all terms r and $I \models r_1 = r_2$ holds iff $I(r_1) = I(r_2)$, i.e., iff r_1 and r_2 are syntactically identical. Obviously, I is a model of $\forall X\ X = X$. Thus, it follows from the precondition that $I \models \sigma(s = t)$ and thus $\sigma(s) = \sigma(t)$.

(b) If $A = p(\ldots)$, $B = q(\ldots)$ with $p \neq q$, then there is no σ such that $\sigma(A)$ and $\sigma(B)$ are syntactically identical. Since $\overline{A = B} = \mathsf{fail}$, we also have $\{\forall X\ X = X, \mathsf{true}\} \not\models \sigma(\overline{A = B})$.

If $A = B = p$, then $\sigma(A)$ and $\sigma(B)$ are syntactically identical for all substitutions σ. Since in this case $\overline{A = B} = \mathsf{true}$, we also have $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A = B})$.

If $A = p(s_1, \ldots, s_n)$ and $B = p(t_1, \ldots, t_n)$, then $\sigma(A) = \sigma(B)$ holds iff $\sigma(s_i) = \sigma(t_i)$ holds for all $1 \leq i \leq n$. According to (a), this is the case iff $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(s_i = t_i)$ holds for all i. So this is equivalent to $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A = B})$.

□

As "$\{\forall X\ X = X, \mathsf{true}\}$" axiomatizes unification, in this way one can of course also determine if two terms are not unifiable. Therefore, a configuration (G_1, CO_1) should only be transformed into a new configuration (G_2, CO_2) if the new condition CO_2 is satisfiable under these axioms, i.e., if $\{\forall X\ X = X, \mathsf{true}\} \models \exists\ CO_2$. This ensures that one does not perform computation steps where unification would fail. For instance, in Example 6.1.12 this would prevent that the first program clause is directly applied for the query $\{\neg\mathsf{add}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), U)\}$. Otherwise, one would then obtain the conjunction $\mathsf{s}(\mathsf{0}) = X \wedge \mathsf{s}(\mathsf{0}) = \mathsf{0} \wedge U = X$. However, we have

$$\{\forall X\ X = X, \mathsf{true}\} \not\models \exists X, U\ \mathsf{s}(\mathsf{0}) = X \wedge \mathsf{s}(\mathsf{0}) = \mathsf{0} \wedge U = X.$$

In order to define the procedural semantics of constraint logic programs, we now extend the idea described in Example 6.1.10 and 6.1.12 as follows: Now the second component CO of a configuration can not only contain constraints with true, fail, and $=$, but also constraints that are constructed with predicate symbols from Δ'. Therefore, the axioms CT have to be considered in addition to the axioms $\forall X\ X = X$ and true when checking the satisfiability of CO and when simplifying CO with "simplify". Here, we assume that there is a solver available to check the validity of existentially quantified constraints. In other words, if CO is a conjunction of constraints, we assume that it is possible to decide whether $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ CO$ holds. (In reality this is of course not the case with constraint theories like CT_{FD}. Thus, in Section 6.2 we will discuss how to "solve" this problem in practice).

Definition 6.1.14 (Procedural Semantics of Constraint Logic Programs)
Let $\mathcal{P}$ be a logic program with constraints and let CT be an associated constraint theory.

- *A* configuration *is a pair (G, CO), where G is a query or the empty clause $\Box$, and where CO is a conjunction of constraints.*
- *There is a* computation step $(G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2)$ *iff $G_1 = \{\neg A_1, \ldots, \neg A_k\}$ with $k \geq 1$ and one of the following two possibilities (A) or (B) holds:*

(A) There is an $1 \leq i \leq k$ such that A_i is not a constraint. Then:

- *there are a program clause $K \in \mathcal{P}$ and a variable renaming ν with $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ and $n \geq 0$, such that*
  - *$\nu(K)$ does not have any variables in common with G_1 or CO_1*
  - *$CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ (CO_1 \wedge \overline{A_i = B})$*
- $G_2 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}$
- $CO_2 = CO_1 \wedge \overline{A_i = B}$

(B) There is an $1 \leq i \leq k$ such that A_i is a constraint. Then:

- $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ CO_1 \wedge A_i$
- $G_2 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg A_{i+1}, \ldots, \neg A_k\}$

– $CO_2 = CO_1 \wedge A_i$

- *A* computation *of $\mathcal{P}$ for the query $G = \{\neg A_1, \ldots, \neg A_k\}$ is a (finite or infinite) sequence of configurations of the form*

$$(G, \mathsf{true}) \vdash_{\mathcal{P}} (G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2) \vdash_{\mathcal{P}} \ldots$$

- *A computation that started with (G, true) and ends in $(\square, CO)$ is* successful. *The computed* answer constraints *are* simplify(CO), *where $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \forall\ (CO \leftrightarrow \text{simplify}(CO))$.*

The procedural semantics *of $\mathcal{P}$ w.r.t. G is defined as*

$$P[\![\mathcal{P}, CT, G]\!] = \{\sigma(A_1 \wedge \ldots \wedge A_k) \mid (G, \mathsf{true}) \vdash_{\mathcal{P}}^{+} (\square, CO), \quad \sigma \text{ is ground substitution with } CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)\}.$$

So there are now two possibilities for a computation step, depending on whether the selected atom A_i from the query is a constraint or not. If it is not a constraint (Case (A)), then SLD resolution with a program clause takes place as before. However, the required unification of A_i and the head B of the program clause is not carried out, but the constraint $\overline{A_i = B}$ are added instead. Here, one should always check whether the collected conjunction of constraints is still satisfiable. However, if A_i is a constraint (Case (B)), then A_i is directly added to the collected conjunction of constraints, provided that it remains satisfiable.

Example 6.1.15 *Consider again the program $\mathcal{P}$ from Example 6.1.6, the constraint theory CT_{FD} from Example 6.1.4, and the query $G = \{\neg\mathsf{fact}(1, Z)\}$. While Example 6.1.8 showed that the declarative semantics results in $D[\![\mathcal{P}, CT, G]\!] = \{\mathsf{fact}(1,1)\}$, we will now illustrate the procedural semantics. Here we apply the function "*simplify*" after each computation step to simplify the obtained conjunction of constraints. In addition, we have omitted the negations in front of the individual literals to increase readability. We get*

$$\begin{aligned}
& (\mathsf{fact}(1, Z), \quad \mathsf{true}) \\
\vdash_{\mathcal{P}}\ & (X \#> 0, X_1 \#= X - 1, \mathsf{fact}(X_1, Y_1), Y \#= X * Y_1, \quad \underbrace{\mathsf{true} \wedge \overline{\mathsf{fact}(1, Z) = \mathsf{fact}(X, Y)}}_{X=1 \wedge Z=Y}) \\
\vdash_{\mathcal{P}}\ & (X_1 \#= X - 1, \mathsf{fact}(X_1, Y_1), Y \#= X * Y_1, \quad \underbrace{X \#> 0 \wedge X = 1 \wedge Z = Y}_{X=1 \wedge Z=Y}) \\
\vdash_{\mathcal{P}}\ & (\mathsf{fact}(X_1, Y_1), Y \#= X * Y_1, \quad \underbrace{X_1 \#= X - 1 \wedge X = 1 \wedge Z = Y}_{X_1=0 \wedge X=1 \wedge Z=Y}) \\
\vdash_{\mathcal{P}}\ & (Y \#= X * Y_1, \quad \underbrace{\overline{\mathsf{fact}(X_1, Y_1) = \mathsf{fact}(0,1)} \wedge X_1 = 0 \wedge X = 1 \wedge Z = Y}_{X_1=0 \wedge Y_1=1 \wedge X=1 \wedge Z=Y}) \\
\vdash_{\mathcal{P}}\ & (\square, \quad \underbrace{Y \#= X * Y_1 \wedge X_1 = 0 \wedge Y_1 = 1 \wedge X = 1 \wedge Z = Y}_{Y=1 \wedge X_1=0 \wedge Y_1=1 \wedge X=1 \wedge Z=1})
\end{aligned}$$

Therefore, the result of the final simplification is the following formula CO:

$$Y = 1 \wedge X_1 = 0 \wedge Y_1 = 1 \wedge X = 1 \wedge Z = 1$$

Thus, the only ground substitution with $CT_{FD} \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ is therefore $\sigma = \{Y/1,\ X_1/0,\ Y_1/1,\ X/1,\ Z/1\}$. Hence, we obtain $P[\![\mathcal{P}, CT_{FD}, G]\!] = \{\mathsf{fact}(1,1)\}$.

Now we can prove the equivalence of the declarative and the procedural semantics for constraint logic programs. This shows that an implementation of constraint logic programming as in Def. 6.1.14 indeed corresponds to the desired (declarative) semantics and that thus (due to Corollary 6.1.9), ordinary logic programming is actually implemented correctly if the constraint theory CT is empty. For the proof of the following theorem we proceed similar to the proof of the corresponding theorem for ordinary logic programs (Theorem 4.1.8).

Theorem 6.1.16 (Equivalence of Declarative and Procedural Semantics) *Let $\mathcal{P}$ be a constraint logic program and CT be an associated constraint theory. Moreover, let $G = \{\neg A_1, \ldots, \neg A_k\}$ be a query. Then we have $D[\![\mathcal{P}, CT, G]\!] = P[\![\mathcal{P}, CT, G]\!]$.*

Proof. We first show the soundness of the procedural semantics w.r.t. the declarative semantics, i.e., $P[\![\mathcal{P}, CT, G]\!] \subseteq D[\![\mathcal{P}, CT, G]\!]$. Let $\sigma(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, CT, G]\!]$. Then there is a successful computation of the form

$$(G, \mathsf{true}) \vdash_{\mathcal{P}} (G_1, CO_1) \vdash_{\mathcal{P}} (G_2, CO_2) \ldots \vdash_{\mathcal{P}} (\Box, CO)$$

with $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$.

We have to show that then we have $\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_k)$. We use induction on the length l of the computation. More precisely, l is the number of $\vdash_{\mathcal{P}}$-steps in the computation. Thus, there is an atom A_i in the query G which is replaced or omitted in the first step $(G, \mathsf{true}) \vdash_{\mathcal{P}} (G_1, CO_1)$.

Case 1: A_i is not a constraint

Then there is a program clause $K \in \mathcal{P}$ and a variable renaming ν with $\nu(K) = \{B, \neg C_1, \ldots, \neg C_n\}$ and $n \geq 0$. Here G and $\nu(K)$ have no common variables, $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\, \overline{A_i = B}$ holds, and we have

$$G_1 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\} \text{ and } CO_1 = \overline{A_i = B}. \tag{6.3}$$

In the induction base, we have $l = 1$. Then $G_1 = \Box$ and thus, $i = k = 1$, $n = 0$, $\nu(K) = \{B\}$ (i.e., the program clause K is a fact), and $CO = CO_1$. Since $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$, we have $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A_1 = B})$ and therefore also $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A_1 = B})$, since CT does not contain the predicate symbols true, fail, and $=$. By Lemma 6.1.13 (b), σ is a unifier of A_1 and B, i.e., $\sigma(A_1) = \sigma(B)$. Since B is also a clause from $\mathcal{P}$, we obtain $\mathcal{P} \models \sigma(B)$ resp. $\mathcal{P} \cup CT \models \sigma(B)$, and therefore also $\mathcal{P} \cup CT \models \sigma(A_1)$.

Now we consider the induction step $l > 1$. For G_1 as in (6.3), we clearly also have the computation

$$(G_1, \mathsf{true}) \vdash_{\mathcal{P}} (G_2, CO_2') \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\Box, CO'),$$

where $CO = \overline{A_i = B} \wedge CO'$ holds. From $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$, we therefore obtain $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO')$. By the induction hypothesis, we have

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge C_1 \wedge \ldots \wedge C_n \wedge A_{i+1} \wedge \ldots \wedge A_k).$$

Since $\mathcal{P} \models C_1 \wedge \ldots \wedge C_n \rightarrow B$ holds, we also have

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge B \wedge A_{i+1} \wedge \ldots \wedge A_k).$$

Furthermore, since $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ implies $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A_i = B})$ (and therefore $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(\overline{A_i = B})$), from Lemma 6.1.13 (b) we obtain $\sigma(A_i) = \sigma(B)$. This results in

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \ldots \wedge A_k).$$

Case 2: A_i is a constraint

Then $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ A_i$ and we have

$$G_1 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg A_{i+1}, \ldots, \neg A_k\} \text{ and } CO_1 = A_i. \tag{6.4}$$

In the induction base, we have $l = 1$. Then we again have $G_1 = \Box$ and therefore $i = k = 1$ and $CO = CO_1$. Since $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ holds, we obtain $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(A_1)$ and therefore also $\mathcal{P} \cup CT \models \sigma(A_1)$, since $\mathcal{P}$ also contains $\{\forall X\ X = X, \mathsf{true}\}$.

Now we consider the induction step $l > 1$. For G_1 as in (6.4) we also have the computation

$$(G_1, \mathsf{true}) \vdash_{\mathcal{P}} (G_2, CO_2') \vdash_{\mathcal{P}} \ldots \vdash_{\mathcal{P}} (\Box, CO'),$$

where $CO = A_i \wedge CO'$. From $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ we obtain $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO')$. By the induction hypothesis, this results in

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge A_{i+1} \wedge \ldots \wedge A_k).$$

Since $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO)$ also implies $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(A_i)$, we have

$$\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \ldots \wedge A_k),$$

since $\mathcal{P}$ also contains $\{\forall X\ X = X, \mathsf{true}\}$.

Now we show the completeness of the procedural semantics w.r.t. the declarative semantics, i.e., $D[\![\mathcal{P}, CT, G]\!] \subseteq P[\![\mathcal{P}, CT, G]\!]$. Let $\sigma(A_1 \wedge \ldots \wedge A_k) \in D[\![\mathcal{P}, CT, G]\!]$. Then $\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_k)$ holds. We now distinguish the individual atoms $A_1, \ldots, A_k$ according to whether they are constraints or not. W.l.o.g. let $A_1, \ldots, A_j$ be no constraints and let $A_{j+1}, \ldots, A_k$ be constraints (where $0 \leq j \leq k$). Since $\mathcal{P}$ does not contain positive literals with predicate symbols from $\Delta' \cup \{\mathsf{true}, \mathsf{fail}, =\}$ (except for the clauses $\{X = X\}$ and $\{\mathsf{true}\}$), and since CT only contains the predicate symbols from Δ', $\mathcal{P} \cup CT \models \sigma(A_1 \wedge \ldots \wedge A_k)$ is equivalent to $\mathcal{P} \models \sigma(A_1 \wedge \ldots \wedge A_j)$ and $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(A_{j+1} \wedge \ldots \wedge A_k)$. As in the completeness proof of the procedural semantics for ordinary logic programs (Theorem 4.1.8) it is shown that $\mathcal{P} \models \sigma(A_1 \wedge \ldots \wedge A_j)$ implies that we have the computation

$$(\{\neg A_1, \ldots, \neg A_j\}, \mathsf{true}) \vdash_{\mathcal{P}}^{+} (\Box, CO_1).$$

Therefore there is also the computation

$$(\{\neg A_1, \ldots, \neg A_k\}, \mathsf{true}) \vdash_{\mathcal{P}}^{+} (\{\neg A_{j+1}, \ldots, \neg A_k\}, CO_1).$$

Here, CO_1 contains the equations between the atoms to be unified. Since σ is a unifier, Lemma 6.1.13 (b) implies that we also have $\{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO_1)$ holds. Moreover, we have

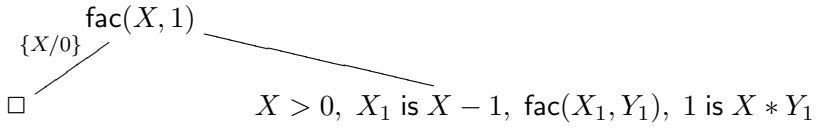
$$(\{\neg A_{j+1}, \ldots, \neg A_k\}, CO_1) \vdash^+_{\mathcal{P}} (\Box, CO_1 \wedge CO_2),$$

where $CO_2 = A_{j+1} \wedge \ldots \wedge A_k$. Since $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(A_{j+1} \wedge \ldots \wedge A_k)$, we now have $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \sigma(CO_1 \wedge CO_2)$. Hence, we obtain $\sigma(A_1 \wedge \ldots \wedge A_k) \in P[\![\mathcal{P}, CT, G]\!]$. □

To solve Indeterminism 2 (i.e., the indeterminism in the selection of the literal from the query), one proceeds again exactly as in ordinary logic programming. So in constraint logic programming, again restrict ourselves to canonical computations, i.e., to computations where always the leftmost literal of the query is chosen. In Def. 6.1.14, we would therefore always have $i = 1$. Indeterminism 1 is again solved by traversing the SLD tree in depth-first search from left to right. This means that program clauses are again considered in the order from top to bottom.

To illustrate the resulting SLD trees, we will again consider the `fac` and the `fact` program from Example 6.1.6.

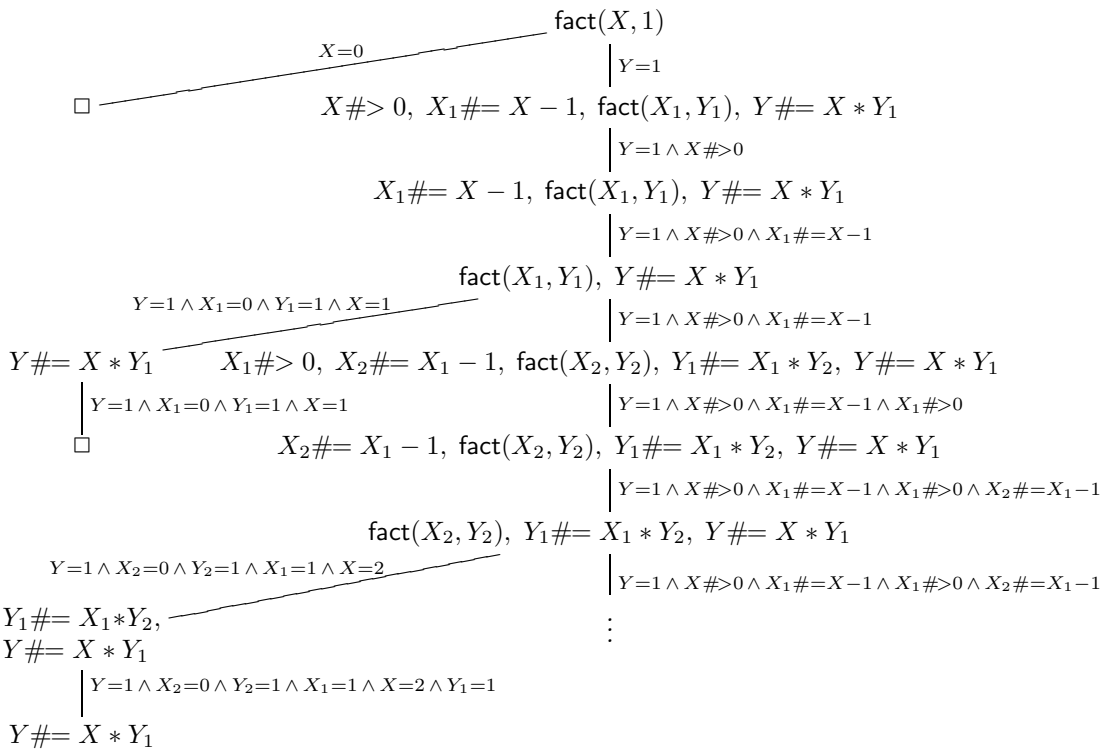
Example 6.1.17 *For the* `fac` *program from Example 6.1.6 and the query "*`?- fac(X,1)`*", we obtain the following SLD tree.*



So the first answer substitution is `X = 0`*. But if one enters "*`;`*" afterwards, this leads to an abort of the program, because the proof goal "*`X > 0`*" is not permitted in* Prolog*. The problem is that the predicate* `>` *expects two fully instantiated arithmetic expressions as arguments. Thus, the program* `fac` *is not bidirectional. It is suitable for computing the factorial of given numbers (i.e., for queries like "*`?- fac(1,Z)`*"), but not for checking for a given number (like* `1`*), whether there is a number* `X` *such that the factorial of* `X` *is* `1`*.*

This demonstrates one of the advantages of constraint logic programming. In contrast to "`>`*", the predicate symbol "*`#>`*" is indeed bidirectional. Therefore, let us now ask the query "*`?- fact(X,1)`*" for the* `fact` *program from Example 6.1.6. For the SLD trees, we will now label the edges with the collected conjunction CO of constraints, simplified using "*simplify*". If the resulting conjunction CO does not satisfy $CT \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\ CO$, then the corresponding child node is not created in the SLD tree. Therefore we obtain the following*

SLD tree:



The rightmost path in the tree is infinite. Here, the problem is that the constraint $Y\#= X*Y_1$ *from the query is never added to the conjunction* CO *of constraints, since the* $\mathsf{fact}(\ldots)$ *constraint always occurs to the left of it. If* $Y\#= X*Y_1$ *were to be added to* CO *instead, the path would immediately result in finite failure, since*

$$Y = 1 \wedge X_1\#= X - 1 \wedge X_1\#> 0$$

implies $Y = 1$ *and* $X\#> 1$*, which contradicts the constraint* $Y\#= X * Y_1$*.*

For a similar reason, the bottommost path with the node "$Y\#= X * Y_1$*" cannot be continued, since "*$Y\#= X * Y_1$*" contradicts the previous constraints*

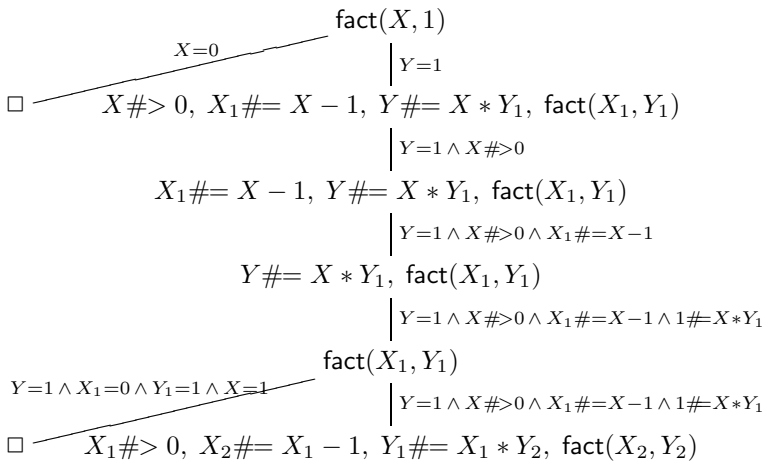
$$Y = 1 \wedge X_2 = 0 \wedge Y_2 = 1 \wedge X_1 = 1 \wedge X = 2 \wedge Y_1 = 1.$$

Thus, if one poses the query "`?- fact(X,1)`*" to the program, then the answers* `X = 0` *and* `X = 1` *are obtained first, since the answer constraints are restricted to those that contain the variable* `X` *of the query. If the user then enters "*`;`*" again, then the program does not terminate anymore.*

To improve this, one should swap the last two atomic formulas in the body of the second `fact` *rule. One then obtains the program*

```
fact(0,1).
fact(X,Y) :- X #> 0, X1 #= X-1,  Y #= X*Y1, fact(X1,Y1).
```

The query "`?- fact(X,1)`*" now leads to the following SLD tree:*



Now the rightmost path cannot be continued any further, because the resulting conjunction of constraints

$$Y = 1 \wedge X\#> 0 \wedge X_1\#= X - 1 \wedge 1\#= X * Y_1 \wedge X_1\#> 0$$

is contradictory. Thus, now the SLD tree is finite and the query "`?- fact(X,1)`*" results in the answers* `X = 0` *and* `X = 1`*, and terminates with "*`false`*" if the user enters "*`;`*" again afterwards.*

6.2 Constraint Logic Programming in Prolog

Now we will show how logic programming with constraints is realized in the programming language Prolog. Note that the predicates "true" and "=" are actually pre-defined with the same clauses as in Def. 6.1.5. Of course, in a Prolog program, one has to specify which CT constraint theory should be used. This also implicitly determines which function and predicate symbols should be contained in the sets Σ' and Δ'.

Like other programming languages, Prolog implementations usually offer a module system. Individual program libraries are then stored in different modules. The predicate `use_module` imports the predicates of a module. For example, to import the library with the predicate symbols of the constraint theory CT_{FD} from Example 6.1.2, the Prolog program must include the following directive :

```
:- use_module(library(clpfd)).
```

Afterwards, the symbols from Σ_{FD} and Δ_{FD} are available (see, e.g., Example 6.1.2) and the underlying constraint theory is CT_{FD}. Thus, one can now actually use the `fact` program from Example 6.1.6 and the answers to the queries "`?- fact(1,Z).`" and "`?- fact(X,1).`" are `Z = 1` resp. `X = 0` and `X = 1`.

But a problem is that entailment from the axioms of constraint theory is often undecidable. For example, for the theory CT_{FD} it is not possible to decide whether

$$CT_{FD} \cup \{\forall X\ X = X, \mathsf{true}\} \models \exists\, CO \tag{6.5}$$

holds for a conjunction of constraints CO. Even for other constraint theories where this question is decidable, the corresponding decision procedure can be too inefficient for a practical application within implementations of constraint logic programming.

For this reason, actual implementations of constraint logic programming usually use techniques that *approximate* the question (6.5). Here, it is possible that the approximation claims that (6.5) holds, although this is not the case. In the case of the constraint theory CT_{FD}, usually one uses the so-called *path consistency*.

Definition 6.2.1 (Path Consistency) *Let $CO = \varphi_1 \wedge \ldots \wedge \varphi_m$ be a conjunction of constraints with $\varphi_i \in \mathcal{A}t(\Sigma_{FD}, \Delta_{FD}, \mathcal{V})$. Let $X_1, \ldots, X_n$ be the variables in CO and let $D_1, \ldots, D_n$ be subsets of $\mathbb{Z}$. We say that $D_1, \ldots, D_n$ are* admissible domains *for the variables $X_1, \ldots, X_n$ w.r.t. CO iff for all constraints φ_i with $1 \leq i \leq m$ and all variables X_j with $1 \leq j \leq n$ we have: For all $a_j \in D_j$ there are $a_1 \in D_1, \ldots, a_{j-1} \in D_{j-1}, a_{j+1} \in D_{j+1}, \ldots, a_n \in D_n$ such that $CT_{FD} \models \varphi_i[X_1/a_1, \ldots, X_n/a_n]$. If there are admissible domains $D_1, \ldots, D_n$ which are all non-empty, then CO is called* path consistent.

The main difference between the path consistency of CO and the "real consistency" (i.e., the question whether $CT_{FD} \models \exists\ CO$ holds) is that path consistency considers the constraints separately. In Prolog implementations, the usual approach is to first set all D_j to be $\mathbb{Z}$. Then, all constraints and variables are successively traversed and the respective domains are reduced. This is done until the domains D_j do not change anymore.

Example 6.2.2 *Consider the following formula CO.*

$$X_1 \#> 5 \ \wedge\ X_1 \#< X_2 \ \wedge\ X_2 \#< 9$$

In the beginning, we have $D_1 = \mathbb{Z}$ and $D_2 = \mathbb{Z}$. We start with regarding the first constraint $X_1 \#> 5$. Now we have to delete all elements from D_1 where this constraint is unsatisfiable. This leads to $D_1 = \{6, 7, \ldots\}$.

Now we regard the second constraint. We start with the variable X_1 and delete all elements $a_1 \in D_1$ for which there is no $a_2 \in D_2$ such that $a_1 < a_2$. However, such elements do not exist. Then we take the variable X_2 and delete all elements $a_2 \in D_2$ for which there is no $a_1 \in D_1$ such that $a_1 < a_2$. This leads to $D_2 = \{7, 8, \ldots\}$.

Now we examine the third constraint. This leads to $D_2 = \{7, 8\}$. This handling of the constraints is repeated until nothing changes anymore. The examination of the second constraint for the variable X_1 results in $D_1 = \{6, 7\}$. Indeed, the query

```
?- X1 #> 5, X1 #< X2, X2 #< 9.
```

results in the answer constraints

```
X1 in 6 ..  7, X1 #< X2, X2 in 7 ..  8
```

This notation with the pre-defined predicate `in` *is equivalent to*

```
6 #=< X1, X1 #=< 7, X1 #< X2, 7 #=< X2, X2 #=< 8
```

The predicate `in` *can also be used by the programmer in programs and queries. Here,* `inf` *and* `sup` *stand for $-\infty$ and ∞. So we could also reformulate our query as follows:*

```
?- X1 in 6 .. sup, X1 #< X2, X2 in inf .. 8.
```

There is also a predicate `label`*, which enforces* Prolog *to enumerate the possible solutions one by one. The argument of* `label` *must be the list of variables whose values one is interested in. Thus, the query*

```
?- X1 #> 5, X1 #< X2, X2 #< 9, label([X1,X2]).
```

results in

```
X1 = 6, X2 = 7 ;
X1 = 6, X2 = 8 ;
X1 = 7, X2 = 8
```

But here the domains for the variables in the argument of `label` *must indeed be finite. Otherwise, the query with* `label` *leads to a program error. For example, this would happen for the following query:*

```
?- X1 #> 5, X1 #< X2, label([X1,X2]).
```

But there are also examples where the constraints are path consistent although they are contradictory.

Example 6.2.3 *The simplest such example is*

```
?- X1 #> X2, X1 #=< X2.
```

The path consistency check shows that for each value $a_1 \in D_1 = \mathbb{Z}$, there is a value $a_2 \in D_2 = \mathbb{Z}$ such that the first constraint is satisfiable and similarly, for each value $a_2 \in D_2 = \mathbb{Z}$, there is a value $a_1 \in D_1 = \mathbb{Z}$ such that the first constraint is satisfiable. The same holds for the second constraint. Thus, the answer is again

```
X1 #> X2, X1 #=< X2.
```

Finally, we want to look at two typical examples of programs where constraint logic programming is particularly useful. The first example again uses the constraint theory CT_{FD}.

Example 6.2.4 *We implement the n-queens problem (our program is essentially from [NM96]). Here, one has a chessboard of size $n \times n$ and the aim is to put n queens on the chessboard which cannot attack each other. This means that there must be no row, column, or diagonal with more than one queen.*

We represent the positions of the queens as a list $[x_1, \ldots, x_n]$, where the number x_i is the number of the row in which the queen of the i-th column is placed. So the positions of the queens are $(x_1, 1), \ldots, (x_n, n)$. Thus, the list $[x_1, \ldots, x_n]$ is a permutation of the numbers $[1, \ldots, n]$. To find out how to put n queens on a $n \times n$ board, one poses the query

```
?- queens(n,L).
```

The atomic formula with the pre-defined predicate `length` *ensures that the result list* `L` *has the length* n*. The predicate* `ins` *from the library* `clpfd` *is similar to* `in`*, but it ensures that* all elements *of the list* `L` *are from the given interval. Thus, "*$[x_1, \ldots, x_n]$ `in 1 ... N`*" holds iff "*x_i `in 1 ... N`*" holds for all* $1 \leq i \leq n$*. The pre-defined predicate* `all_distinct` *from the library* `clpfd` *ensures that all elements of* `L` *are pairwise different. Thus, the* `ins` *and* `all_distinct` *formulas are again abbreviations for (obvious) conjunctions of constraints.*

```
queens(N,L) :- length(L, N),
               L ins 1 .. N,
               all_distinct(L),
               safe(L),
               label(L).

safe([]).
safe([X|Xs]) :- safe_between(X, Xs, 1),
                safe(Xs).

safe_between(X, [], M).
safe_between(X, [Y|Ys], M) :- no_attack(X, Y, M),
                              M1 #= M + 1,
                              safe_between(X, Ys, M1).

no_attack(X, Y, N) :- X+N #\= Y, X-N #\= Y.
```

The predicate `safe` *ensures that the queens in the list* `L` *are not on the same diagonals. We will explain it in more detail below. Finally,* `label(L)` *is used to return the individual elements of* `L`*.*

The atomic formula `safe(L)` *ensures that no queen in* `L` *can attack a queen on a position* to her right*. Here, we use the auxiliary predicate* `safe_between`*, where* `safe_between(X, L, M)` *is true if the queen in line* `X` *cannot attack a queen in the columns from* `L`*, provided that* `M` *is the distance of the column from* `X` *to the first column from* `L`*. Therefore,* `no_attack(X, Y, N)` *holds iff the queen in row* `X` *and the queen in row* `Y`*, which is* `N` *columns further to the right, are not on the same diagonal. For example, one gets*

```
?- queens(4,L).

L = [2, 4, 1, 3] ;
L = [3, 1, 4, 2]
```

In the next example, we consider a different constraint theory for constraints over real numbers.

Example 6.2.5 *We now consider a constraint signature* $(\Sigma, \Delta, \Sigma', \Delta')$ *where* Σ' *and* Δ' *contain function and predicate symbols for operations on real numbers. To distinguish these symbols from the (partly identical) symbols from* $\Sigma \setminus \Sigma'$ *and* $\Delta \setminus \Delta'$*, this time we do not use new names like "#>", but we write all constraints with predicates from* Δ' *in curly brackets. The constraint theory* CT_R *then contains all true formulas over* $\mathbb{R}$*. To import the corresponding library, one needs the directive*

```
:- use_module(library(clpr)).
```

As an example, we now implement a program that can be used to compute interest and repayment of loans [FA10]. Here `mortgage(D,T,I,R,S)` *is true iff*

- `D` *is the amount one has taken out as a loan, i.e., the debt*
- `T` *is the duration (in months) since one took out the loan*
- `I` *is the interest rate one has to pay per month*
- `R` *is the repayment one has to make per month*
- `S` *is the amount of debt one still has after* `T` *months*

Then we obtain the following program:

```
mortgage(D, T, I, R, S) :- {T = 0, D = S}.
mortgage(D, T, I, R, S) :- {T > 0, T1 = T - 1, D1 = D + D * I - R},
                           mortgage(D1, T1, I, R, S).
```

If a month has not yet passed, then the remaining debt `S` *is just the amount* `D` *that was taken out as a loan. Otherwise, due to the interest per month, the debt increases by* `D * I` *and it decreases by the repayment* `R` *every month.*

For instance, one can now ask how large the debt will still be after 30 years (i.e., 360 months) if one originally took out a loan of 100000 Euro, the interest rate per month is 1 % and one pays back 1025 Euro every month.

```
?- mortgage(100000, 360, 0.01, 1025, S).

S = 12625.9
```

*One sees that although one has already paid back 360 * 1025 = 369000 Euro in the meantime, one still has a remaining debt of 12625.9 Euro. This demonstrates the effect of the interest.*

Because of the bidirectionality of logic programming (and the fact that in contrast to the built-in arithmetic predicates, the predicates in the constraints indeed work in a bidirectional way), one can provide the values for arbitrary arguments in the queries, and let Prolog compute the values for the other arguments. For example, one can ask how high the loan should be such that one can pay it off within 30 years with these conditions.

```
?- mortgage(D, 360, 0.01, 1025, 0).

D = 99648.8
```

To ask how long we still have to pay back our original loan of 100000 Euro, one could ask the following query:

```
?- {S =< 0}, mortgage(100000, T, 0.01, 1025, S).

S = -807.964, T = 374.0 ;
S = -1841.04, T = 375.0 ;
S = -2884.45, T = 376.0 ;
S = -3938.3,  T = 377.0 ;
...
```

This means that after 374 months, we would have a "debt" of -807.964 Euro. So in the last month, one does not have to pay back the full rate of 1025 Euro. (The further answers are correct, but not really useful).

These examples illustrate that extending logic programming by constraints is indeed a very reasonable and useful extension of logic programming.

Bibliography

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

[Bra11] I. Bratko. *Prolog Programming for Artificial Intelligence*, Addison-Wesley, 2011.

[CM13] W. F. Clocksin and C. S. Mellish. *Programming in* Prolog, Springer, 2013.

[FA10] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*, Springer, 2010.

[Gie11] J. Giesl. *Termersetzungssysteme*, Course Notes, RWTH Aachen University, 4. Edition, 2011.

[Gie14] J. Giesl. *Funktionale Programmierung*, Course Notes, RWTH Aachen University, 10. Edition, 2021.

[Gra11] E. Grädel. *Mathematische Logik*, Course Notes, RWTH Aachen University, 2011.

[Han87] M. Hanus. *Problemlösen mit* Prolog, Teubner, 1987.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[Llo13] J. W. Lloyd. *Foundations of Logic Programming*, Springer, 2013.

[MS98] K. Marriott and P. J. Stuckey. *Programming with Constraints*, MIT Press, 1998.

[NM96] U. Nilsson and J. Maluszynski. *Logic, Programming and* Prolog, Wiley, 1996.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23-41, 1965.

[Sch92] P. H. Schmitt. *Theorie der logischen Programmierung*, Springer, 1992.

[Sch00] U. Schöning. *Logik für Informatiker*, Spektrum Akademischer Verlag, 2000.

[Sch08] U. Schöning. *Theoretische Informatik – kurzgefasst*, Spektrum Akademischer Verlag, 2008.

[SS00] L. Sterling and E. Shapiro. *The Art of* Prolog, MIT Press, 2000.

Index