Course Material:
Introduction to Automated Reasoning,
Course given at the 18th European Summerschool
in Logic, Language and Information,
Malaga, Spain,
July 31st until August 4th, 2006

Hans de Nivelle and Marc Bezem

June 26, 2006

Contents

Chapter 1

Preliminaries

In this chapter we give some preliminaries. The main intention is to introduce some notation. The reader can take the risk and skip this chapter if he is in a hurry.

1.1 Orders and Ordinal numbers

In this section we define the concepts of order, well-order, and ordinal number. An order is a relation that specifies how to line up the elements in a set:

Definition 1.1.1 *An* order $\prec$ *is a relation with the following properties:*

O1 *Never* $d \prec d$.

O2 *If* $d_1 \prec d_2,$ *and* $d_2 \prec d_3,$ *then* $d_1 \prec d_3$.

An order $\prec$ *is called* total *on a set* S *if*

O3 *For all* $d_1, d_2 \in S,$ *either* $d_1 = d_2, d_1 \prec d_2,$ *or* $d_2 \prec d_1$.

It is possible to obtain a relation $\preceq$ *from* $\prec$ *as follows:*
$d_1 \preceq d_2$ *iff* $d_1 \prec d_2,$ *or* $d_1 = d_2$. *This relation* $\preceq$ *satisfies:*

O1' *Always* $d \preceq d$.

O2' *If* $d_1 \preceq d_2,$ *and* $d_2 \preceq d_3,$ *then* $d_1 \preceq d_3$.

O3' *If* $d_1 \preceq d_2,$ *and* $d_2 \preceq d_1,$ *then* $d_1 = d_2$.

It is also possible to reverse the process: If $\preceq$ is a relation that satisfies O1', O2', and O3', then a relation $\prec$ can be extracted by

$$d_1 \prec d_2 \text{ iff } d_1 \preceq d_2, \text{ and } d_1 \neq d_2.$$

It can be shown that $\prec$ satisfies O1 and O2. As a consequence we will call both $\prec$ and $\preceq$ an order, and use the one that is most convenient, dependent on the situation. In the same way we will use the pair $<, \leq$, and $\sqsubset, \sqsubseteq$.

Definition 1.1.2 *If $\prec$ is an order, and D is a set, and $d \in D$, then d is* maximal *in D if for no $d' \in D$ it is the case that $d \prec d'$. d is minimal in D if for no $d' \in D$, it is the case that $d' \prec d$. If a minimal element is guaranteed to exist and unique, we will write* $\mathrm{Min}(D)$ *for the minimal element of D. In the same way we write* $\mathrm{Max}(D)$ *for the maximal element of D.*

Minimal and maximal elements do not necessarily exist in a set D, and they are not necessarily unique if they exist. If however $\prec$ is total, then maximal and minimal elements are unique if they exist.
An order for which minimal elements do always exist, is called *well-founded*:

Definition 1.1.3 *An order $\prec$ is* well-founded *on a set D, if every nonempty $X \subseteq D$ contains minimal elements. If $\prec$ is total, then $\prec$ is called a* well-order.

It is possible to define a notion of number, which makes it possible to attach a length to each well-order. This type of number is called an *ordinal number*. The class of ordinal numbers has the following properties:

1. 0 is an ordinal number.
2. If α is an ordinal number, then $\alpha + 1$ is an ordinal number.
3. For every set S of ordinal numbers, there is an ordinal number α, which is the smallest ordinal greater than every element in S.

Definition 1.1.4 *We call the class of ordinal numbers* ORD. *We use $<$ for the relation 'is less than'.*

If X is a set of ordinals, then the smallest ordinal which is greater than every element in X, is called the supremum *of X, which is written as* $\mathrm{Sup}(X)$.
The maximal element *of X, notation* $\mathrm{Max}(X)$ *is defined as the smallest ordinal n such that for all $x \in X$, we have $x \leq n$.*
An ordinal is called a successor ordinal *if it can be written as* $\mathrm{Sup}(\{n\})$, *for an ordinal n. The other ordinals are called* limit ordinals.

We have $\mathrm{Sup}(\emptyset) = 0, \mathrm{Sup}(\{n\} = n + 1$, $\mathrm{Sup}(\{0, 1, 2, 3, 4, \ldots\}) = \omega$. ω is a limit ordinal. $1, 2, 3, \ldots$, are successor ordinals.
If X is a finite set of ordinals, then $\mathrm{Max}(X) \in X$.

Definition 1.1.5 *Let D be a set, let $\prec$ be a well-founded relation on D. We define the* rank *of d recursively as follows:*

$$\mathrm{rank}(d) = \mathrm{Sup}(\{\mathrm{rank}(d') \mid d' \prec d\}).$$

We will often make recursive constructions using ordinal numbers: This is done as follows:

- Specify I_0.
- Specify how I_{i+1} can be obtained from I_i.

- Specify, for a limit ordinal i, how I_i can be obtained from the I_λ with $\lambda < i$.

Because every ordinal can be reached using these constructions, this will specify a value for each ordinal.

1.2 Multisets

In the superposition calculus, clauses will be represented by multisets.

Definition 1.2.1 *A multiset S is similar to a set, but it is able to distinguish how often an element occurs in it. The* characteristic function χ_S *is defined by:*

- $\chi_S(x) = 0$ *if x does not occur in S.*
- $\chi_S(x) = n$ *if x occurs n times in S.*

We write [ ] for the empty multiset, $[1, 1, 2]$ is the multiset in which 1 occurs twice, 2 occurs once, and in which no other elements occur. The definition of maximal, and minimal element are the same as for normal (i.e. non multi-) sets. We define the following:

Definition 1.2.2 *If S_1 and S_2 are multisets then*

1. *$S_1 \cup S_2$ is defined by*

$$\chi_{S_1 \cup S_2}(x) = \chi_{S_1}(x) + \chi_{S_2}(x).$$

2. *$S_1 \backslash S_2$ is defined by*

$$\chi_{S_1 \backslash S_2}(x) = 0 \text{ iff } \chi_{S_1}(x) \leq \chi_{S_2}(x),$$

$$\chi_{S_1 \backslash S_2}(x) = \chi_{S_1}(x) - \chi_{S_2}(x), \text{ otherwise.}$$

3. *$S_1 \subseteq S_2$ is defined by*

$$\chi_{S_1}(x) \subseteq \chi_{S_2}(x) \text{ iff } \chi_{S_1}(x) \leq \chi_{S_2}(x), \text{ for all } x.$$

4. *For any $n \in \mathcal{N}$, the set $n.S_1$ is defined by*

$$\chi_{n.S_1}(x) = n.\chi_{S_1}(x).$$

5. *S_1 and S_2 are disjoint if nowhere*

$$\chi_{S_1}(x) \neq 0, \text{ and } \chi_{S_2}(x) \neq 0.$$

Chapter 2

Introduction to Resolution

In this chapter we introduce first order predicate logic. After that we prove some fundamental things about predicate logic, which are used in automated theorem proving. These are Herbrand theorem, and the possibility to Skolemise a formula. We end this chapter by defining unification, and resolution.

2.1 First order languages

Definition 2.1.1 *We assume that we have an infinite set of* variables. *They will be written starting with uppercase letters.*
(e.g. $X, Y, Var1, X1, \ldots$ *) There is a also a set of* constant symbols. *These will be written starting with a lowercase letter. We also consider numbers as constant symbols. Finally we assume that there is a set of* function symbols. *These will be written starting with a lowercase letter. We also consider the standard operator names, like* $+$, *and* $\times$ *as function symbols. Every function symbol has an integer attached to it, which is called the* arity *of the function symbol. The arity indicates the number of arguments that the function expects. We define the* set of terms *as the set of objects that can be constructed in finite time by the following rules:*

- *Each variable name is a term.*
- *Each constant symbol is a term.*
- *If* $t_1, \ldots, t_n$, *with* $n > 0$, *are terms, and* f *is a function symbol, with arity* n, *then* $f(t_1, \ldots, t_n)$ *is a term.*

Terms will be used in first order logic, to indicate objects. Terms are the names of objects. If we have a proposition: 'The grass is green', 'the grass' functions as a term. Something is said about the grass, namely that it is green. We could have also said this about another term, for example 'the sky is green'. The last part of the sentence, '... is green' expects a term and will transform it into a sentence which is true, or not. Such a structure is called a *predicate*. There are

also predicates which need two terms. For example: 'the grass is greener than the sky'. Here both 'the grass' and 'the sky' are terms, and '... is greener than ...' is a predicate which needs two terms. In the same manner as for a function, we will call the number of terms, that a predicate expects the *arity*.
Now if we have two propositions 'the grass is green', and 'the sky is blue', we might like to combine them: 'the grass is green and the sky is blue', or 'the grass is green or the sky is blue'. For a combination of the 'and'-type it is necessary that both components are true. For a proposition of the 'or'-type it is sufficient that one component is true. Whether or not such a combined proposition is true, can be obtained from the truth-values of the components. It is also possible to deny propositions: 'the grass is not green'. Such a denial is not true if the proposition is true. The following combinator is not so innocent: 'if the grass is green then the sky is blue'. This sentence somehow suggests a causal relation between the fact that the grass is green and the fact that the sky is blue. The problem with this causal relation is that it is extremely difficult to formalise, and that it doesn't fit well with the other operators. For this reason a large simplification has been made: A sentence of type 'if X then Y' is true if either X is not true, or Y is true. With this definition, the sentence above is true. Also true is the sentence 'if Paris is the capital of the Netherlands, then Amsterdam is the capital of France'. We do not claim that this last definition is the real meaning of 'if ... then ...'. It is only a convenient simplification.
However in some situations this definition catches the meaning of 'if ... then ...' quite well: The sentence 'if you want to live long you will have to eat healthy food' means that either you can accept a short life or healthy food.
There is another type of sentence that can be formed. If the grass green, then there exists a green thing. 'There is something which is green'. Its counterpart is: 'all things are green'. We will formalise this in the following:

Definition 2.1.2 *We assume that we also have* predicate symbols*. These will be indicated by names, starting with a lowercase letter. Some will also be indicated by special symbols, like* $=$*, or* $\leq$ *. We define formulae:*

- *If* p *is a predicate symbol, with arity* $n \geq 0$*, and* $t_1, \ldots, t_n$ *are terms, then* $p(t_1, \ldots, t_n)$ *is an atom.*

Then:

- *Every atom is a formula.*
- *If* A *is a formula then* $\neg\ A$ *is a formula.*
- *If* A *and* B *are formulae, then so are* $A \vee B$*,* $A \wedge B$*,* $A \rightarrow B$*.*
- *If* A *is a formula, and* X *is a variable, then* $\forall X A$*, and* $\exists X A$*, are formulae.*

2.2 Semantics of first order logic

We define interpretations of first order formulae:

Definition 2.2.1 *Let Γ be a set of first order formulae. An interpretation I of Γ is an ordered tuple $I = (D, [\ ])$, where*

- *D is a nonempty domain,*
- *$[\ ]$ is a function, which attaches*
  - *to every function symbol f in Γ, occurring with arity n, a total function $D^n \rightarrow D$. We write $[f]$ for this function.*
  - *to every predicate symbol p, occurring in Γ, with arity n a subset of D^n. These are the tuples for which the relation holds. We write $[p]$ for this subset.*
  - *to every variable V, which is free in an $F \in \Gamma$, an element of D. We will write $[V]$ for this element.*

Now if we have an interpretation $I = (D, [\ ])$ we can recursively attach a meaning to every term, when the variables are defined by $[\ ]$:

- *If the term is a variable V, then $[V]$ is already defined.*
- *For a constant symbol c, $[c]$ is already defined.*
- *If the term has form $f(t_1, \dots, t_n)$, and f has arity $n > 0$, then $[f(t_1, \dots, t_n)] = [f]([t_1], \dots, [t_n])$.*

We want to define when an interpretation makes a formula true. For this we sometimes need to change $[\ ]$ a little: $[\ ]^V_d$ is obtained from $[\ ]$ by changing the value of V into d, if $[V]$ was defined, or adding it if $[V]$ was undefined:

1. *$[\ ]^V_d$ has the same value as $[\ ]$, for all functions and constant symbols.*
2. *For all variables X different from V, $[X]^V_d$ is defined if and only if $[X]$ is defined. If $[X]^V_d$ and $[X]$ are both defined they have the same value.*
3. *$[V]^V_d = d$, independent of whether or not $[V]$ was defined.*

Next we define when a formula is true or not in the interpretation. Let F be a formula, let I be an interpretation of F.

- *If F is an atom, then F has form $F = p(t_1, \dots, t_n)$. Here n is the arity of p. We define that $p(t_1, \dots, t_n)$ is true in I if $([t_1], \dots, [t_n]) \in [p]$. Otherwise $p(t_1, \dots, t_n)$ is false in I.*
- *If F is of the form $\neg\, A$, then F is true in I if A is false in I. F is false in I if A is true in I.*
- *If F is of the form $A \vee B$ then F is true in I if either A is true, or B is true in I. Otherwise F is false in I.*
- *If F is of the form $A \wedge B$, then F is true in I if both A and B are true in I. Otherwise F is false in I.*

- *If F is of the form $A \rightarrow B$, then F is true if either A is not true in I, or B is true in I. Otherwise F is false in I.*
- *If F has form $\forall V A$, then F is true in $I = (D, [\ ])$ if A is true in $I' = (D, [\ ]_d^V)$, for every $d \in D$. Otherwise F is false in I.*
- *If F has form $\exists V A$, then F is true in $I = (D, [\ ])$ if A is true in $I' = (D, [\ ]_d^V)$, for a $d \in D$. Otherwise F is false in I.*

We say that a formula (or a set of formulae) is satisfiable *if there is an interpretation in which this formula, (or every formula in the set) is true.*

2.3 Sequent Calculus

In this section we give a *deduction system* for first order logic. The system that we define here is a sequent calculus. Sequent calculus is not completely standard. In most deduction systems formulae are added to a set, according to certain rules. If one wants to prove that Γ implies A, one starts with Γ, and tries to obtain the A, using the rules. For example there may be rules:

arrow If A and $A \rightarrow B$ are derived, then it is possible to derive B.

or From A derive $A \vee B$ and $B \vee A$.

and From A and B derive $A \wedge B$.

Sequent calculus works differently. Instead of deriving consequences, complete implications are derived. So the rule: From A and B the formula $A \wedge B$ can be derived, is replaced by the rule: If Γ implies A, and Γ implies B, then Γ implies $A \wedge B$. The first method has the disadvantage that it disturbs the symmetry of first order logic, by making a difference between assumption, and conclusion. In sequent calculus it is not necessary to make this distinction, and the symmetry of classical logic is preserved in the system. Before we can define sequent calculus, we need the following notions:

Definition 2.3.1 *Let F_1 and F_2 be formulae. We call F_1* a renaming *of F_2 if the following two things are the case:*

1. *F_1 and F_2 have the same structure. This means that if all variables in F_1 and F_2 are replaced by one variable, say X, then F_1 and F_2 are equal. This implies that F_1 and F_2 have the same structure, and that F_1 has a variable at a certain position if and only if F_2 has a variable at this position.*
2. *If X is a variable occurring in F_1 at a certain position, and Y occurs in F_2 at the same position, then either*
   (a) *X is free in F_1 and Y is free in F_2, and $X = Y$.*
   (b) *X and Y are both bound a quantifier. This quantifier must occur at the same position in both F_1 and F_2, and it must be universal or existential in both formulae.*

This is a fairly complicated definition. We will consider formulae which are renamings of each other, equal.

Example 2.3.2 *The following pairs of formulae are renamings of each other:*

$p(X,Y)$	$p(X,Y)$
$\forall X \forall Y p(X,Y)$	$\forall Y \forall X p(Y,X)$
$\forall X \forall Y p(X)$	$\forall Y \forall Y p(Y)$

Definition 2.3.3 *Let F be a formula, let X be a variable. A* free occurrence *of X in F, is a position in F which contains X, and which is not in the scope of a quantifier $\forall X$ or $\exists X$. Variable X is* free *in F if X has a free occurrence in F. These notions are well-defined because they are invariant under renaming. Let F be a formula, let X be a variable, let t be a term. The formula $F[X := t]$ is obtained by*

1. *selecting a renaming of F, for which each free occurrence of X is not in the scope of a quantifier which binds a variable occurring in t.*
2. *replacing all free occurrences of X by t in this renaming.*

We will call this replacement substitution*. The formula $F[X := t]$ is called a* substitution instance *of F, (or only instance)*

It can be easily checked that when different renamings are chosen in (1), the results from the substitution will be renamings of each other.
Substitution will be used for the following: If one has proven a general thing, for example $\forall X \forall Y p(X,Y)$ then this implies $p(1,2)$ and $p(s(s(X)), s(Y))$.

Definition 2.3.4 *A* sequent *is a structure of the form*

$$\Gamma \vdash \Delta.$$

The set Γ is a finite set of formulae, possibly empty. If $\Gamma = \emptyset$, we write $\vdash \Delta$. The set Δ is a finite set of formulae, possibly empty. If $\Delta = \emptyset$, we write $\Gamma \vdash$. If both Γ and Δ are empty, then we write $\vdash$.
We say that a sequent $\Gamma \vdash \Delta$ holds *if every interpretation $I = (D, [\ ])$, of Γ, Δ, which makes all Γ true, makes one of the Δ true.*
A sequent $\Gamma \vdash \Delta$ is true in an interpretation $I = (D, [\ ])$ if either

1. *I does not make all Γ true, or*
2. *I makes one of the Δ true.*

Next we give the replacement rules. The bars ———— mean that the sequent under the bar may be added if the sequents above the bar are present.

Definition 2.3.5 *The rules for sequent calculus:*

Axiom

$$\frac{}{A \vdash A}$$

Structural Rules

$$Kl\text{:}\ \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \qquad Kr\text{:}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, B}$$

Logical Rules

$$\wedge l\text{:}\ \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad \wedge r\text{:}\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$$

$$\vee l\text{:}\ \frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \qquad \vee r\text{:}\ \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}$$

$$\rightarrow l\text{:}\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma, B \vdash \Delta}{\Gamma, A \rightarrow B \vdash \Delta} \qquad \rightarrow r\text{:}\ \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \rightarrow B}$$

$$\neg l\text{:}\ \frac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta} \qquad \neg r\text{:}\ \frac{\Gamma, B \vdash \Delta}{\Gamma \vdash \Delta, \neg B}$$

Quantifier Rules

$$\forall l\text{:}\ \frac{\Gamma, A[X := t] \vdash \Delta}{\Gamma, \forall X A \vdash \Delta} \qquad \forall r\text{:}\ \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, \forall X A}$$

$$\exists l\text{:}\ \frac{\Gamma, A \vdash \Delta}{\Gamma, \exists X A \vdash \Delta} \qquad \exists r\text{:}\ \frac{\Gamma \vdash \Delta, A[X := t]}{\Gamma \vdash \Delta, \exists X A}$$

In the rules $\forall r$, and $\exists l$, there is the condition that X is not free in Γ or Δ.

Theorem 2.3.6 *The sequent calculus given in Definition 2.3.5 is sound and complete. That means: A sequent $\Gamma \vdash \Delta$ is derivable if and only if it holds.*

A proof can be found in ([Gal86]).
In this sequent calculus the Γ and Δ of a sequent are sets, and this is very natural, because of the meaning of a sequent. There is no point in distinguishing how often a hypothesis is assumed, or how often a consequence is derived.
It is however possible to restrict this identification of the sequents $\Gamma \vdash \Delta, A, A$ and $\Gamma \vdash \Delta, A$. In a sequent $\Gamma \vdash \Delta$, it is possible to take Γ and Δ as multisets instead of sets, (or even lists). In that case some rules that are implicit when the Γ and Δ are sets, become explicit:

- From $\Gamma, A, A \vdash \Delta$ derive $\Gamma, A \vdash \Delta$,
- from $\Gamma \vdash \Delta, B, B$ derive $\Gamma \vdash \Delta, B$.

These rules are called *contraction rules*. Rules for the other direction are not necessary because they are captured by the rules Kl and Kr. In the case that Γ and Δ are lists, it is also meaningful to define a permutation rule.
These rules, which restructure sequents rather than derive them, are called *structural rules*. The weakening rule is also called a structural rule. Logic which have a sequent calculus in which these structural rules are in one or other form restricted are called *substructural logics*. An example of such a logic is *intuitionistic logic*, in which Γ is a set, and Δ contains at most one element. This makes contraction on the right impossible.
Another example is *linear logic*, the logic in which Γ and Δ are multisets, and where weakening and contraction are absent. It has two ands, and two ors. Unfortunately it is rather difficult to give a meaning to sequents of this logic, but it has many interesting proof theoretic properties.
There is one special rule which we did not mention here, the so called *cut rule*. It is the following rule:

Cut rule

$$\frac{\Gamma_1, A \vdash \Delta_1 \qquad \Gamma_2 \vdash \Delta_2, A}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}$$

This rule is a redundant rule, because there exists a method to remove this rule from a proof if it is used. If this rule occurs in a proof it is possible to push it downward until it is no longer present. This process is called *cut elimination*. The cut rule has an undesirable property, which the other rules do not have, namely that the formula A completely dissapears. In all other rules, the formulae in the parent sequents are present as subformulae in the derived sequent. This property is called the *subformula* property.

2.4 Skolemisation

In this section we define Skolemisation. This technique makes it possible to remove some quantifiers from a sequent that one wants to prove. Suppose that one wants to prove $\vdash \forall X p(X)$. Then it is sufficient to prove this sequent for an arbitrary X. It is possible to give this X a name c. In this manner the sequent can be replaced by $\vdash p(c)$. We will define Skolemisation in general. After that we prove that a sequent holds if and only if its Skolemisation holds.

Definition 2.4.1 *We attach a* polarity *to all occurrences of (sub)formulae of a sequent. Let $\Gamma \vdash \Delta$ be a sequent. We say that all formulae in Γ occur* negatively *in the sequent, and that all formulae in Δ occur* positively *in the sequent. Furthermore:*

- *if a formula $A \vee B$, or $A \wedge B$ occurs positively in the sequent, then A and B occur positively in the sequent.*
- *if a formula $A \vee B$, or $A \wedge B$ occurs negatively in the sequent, then A and B occur negatively in the sequent.*

- *if a formula $A \rightarrow B$ occurs positively in the sequent then A occurs negatively in the sequent, and B occurs positively in the sequent.*
- *if a formula $A \rightarrow B$ occurs negatively in the sequent, then A occurs positively in the sequent, and B occurs negatively in the sequent.*
- *if a formula $\neg F$ occurs positively in the sequent, then F occurs negatively in the sequent.*
- *if a formula $\neg F$ occurs negatively in the sequent, then F occurs positively in the sequent.*
- *if a formula $\forall X F$, or $\exists X F$ occurs positively in the sequent, then F occurs positively in the sequent.*
- *if a formula $\forall X F$, or $\exists X F$ occurs negatively in the sequent, then F occurs negatively in the sequent.*

This makes it possible to attach a polarity to all subformulae. (on a given position in a sequent)

Using this we can define Skolemisation.

Definition 2.4.2 *Let $\Gamma \vdash \Delta$ be a sequent. A* Skolemisation *of $\Gamma \vdash \Delta$ is obtained by iteratively:*

1. *Selecting an outermost positive occurrence of a $\forall$-quantifier, or an outermost negative occurrence of an $\exists$-quantifier, (outermost means: Not in the scope of another positively occurring $\forall$, or negatively occurring $\exists$) and writing $F[QXA]$, for the formula in which the quantifier was found.*
2. *Collecting all variables in A that are bound by a quantifier (which must be a positively occurring $\exists$, or negatively occurring $\forall$) outside A. Write $V_1, \ldots, V_n$ for all these variables. It is possible that $n = 0$.*
3. *Replacing all occurrences of X by $f(V_1, \ldots, V_n)$, where f is a new function symbol which did not occur in $F[A]$ before,*

This is done until no such replacements are possible. The functions f are called the Skolem functions*. The terms $f(V_1, \ldots, V_n)$ are called* Skolem terms.

Skolemisations are independent of renamings.

Theorem 2.4.3 *Let $\Gamma \vdash \Delta$ be a sequent. Let $\Gamma' \vdash \Delta'$ be its Skolemisation. We have: $\Gamma \vdash \Delta$ holds iff $\Gamma' \vdash \Delta'$ holds.*

The proof is far from easy. We give only a sketch.
If $\Gamma \vdash \Delta$ is a sequent then it is possible to find a renaming such that all quantifiers have a different variable. In the following it is assumed that this is done in a fixed manner. The variables that will be replaced by Skolem terms will be

denoted as $W_1, \ldots, W_s$. The chosen Skolem terms will be $f_1(V_{1,1}, \ldots, V_{1,l_1}), \ldots, f_s(V_{s,1}, \ldots, V_{s,l_s})$. It is also assumed that these are fixed. With $l_1, \ldots, l_s$ we will denote the arities of the Skolem functions. We will adopt some more notation: If $(D, [\,])$ is an interpretation of $\Gamma \vdash \Delta$, we write $[\,](d_{i,1}, \ldots d_{i,l_i})$ instead of $(\cdots([\,]^{V_{i,1}}_{d_{i,1}})^{V_{i,2}}_{d_{i,2}} \cdots)^{V_{l_i}}_{d_{i,l_i}}$, for obvious reasons. We write $[\,](d_{i,1}, \ldots, d_{i,l_i}; d)$ instead of $[\,](d_{i,1}, \ldots, d_{i,l_i})^{W_i}_d$.

Definition 2.4.4 *Let $\Gamma \vdash \Delta$ be a sequent. Let $I = (D, [\,])$ be an interpretation of $\Gamma \vdash \Delta$. We define* Skolem relations *belonging to $\Gamma \vdash \Delta$ and I as relations $P_1, \ldots P_s$ on D with arities $l_1 + 1, \ldots, l_s + 1$. The i-th Skolem relation belongs to the i-th Skolem function. They must satisfy the following condition:*

- *If P_i belongs to a formula QW_iA, and the Skolem term belonging to W_i is $f_i(V_{i,1}, \ldots, V_{i,l_i})$, then: For every $d_{i,1}, \ldots, d_{l_i} \in D$, for every x and y, such that $P_i(d_{i,1}, \ldots, d_{l_i}, x)$ and $P_i(d_{i,1}, \ldots, d_{l_i}, y)$ it must be that*
    - *D and $[\,](d_{i,1}, \ldots, d_{i,l_i}; x)$ makes A true iff*
    - *D and $[\,](d_{i,1}, \ldots, d_{i,l_i}; y)$ makes A true.*

  This definition is meaningful, because the truth value of A is determined by D and $[\,](d_{i,1}, \ldots, d_{i,l_i}; x)$.

- *For all $d_{i,1}, \ldots, d_{i,l_i} \in D$, there exists at least one $d \in D$, such that $P_i(d_{i,1}, \ldots, d_{i,l_i}, d)$.*

Definition 2.4.5 *Let $\Gamma \vdash \Delta$ be a sequent. Let $I = (D, [\,])$ be an interpretation of $\Gamma \vdash \Delta$. Let $P_1, \ldots, P_s$ be Skolem relations. We define when I* makes a formula true with *$P_1, \ldots, P_s$. It is defined by cases, like in Definition 2.2.1. All cases are the same, except the following:*

- *if F has form $\forall W_iA$, and F is positively occurring in $\Gamma \vdash \Delta$, then F is true in I with $P_1, \ldots, P_s$ if A is true in every $I' = (D, [\,]^V_d)$, such that $P_i(d_{i,1}, \ldots, d_{i,l_i}, d)$ Here the $d_{i,j}$ are the assignments to the $V_{i,j}$, so $d_{i,1} = [V_{i,1}], \ldots, d_{i,l_i} = [V_{i,l_i}]$. Otherwise F is false in I with $P_1, \ldots, P_s$.*

- *if F has form $\exists W_iA$, and F is negatively occurring in $\Gamma \vdash \Delta$, then F is true in I with $P_1, \ldots, P_s$ if there is a $d \in D$, such that $P_i(d_{i,1}, \ldots, d_{i,l_i}, d)$, and $I' = (D, [\,]^{W_i}_d)$ makes A true. Here $d_{i,1} = [V_{i,1}], \ldots, d_{i,l_i} = [V_{i,l_i}]$.*

- *If F is of the form $\forall VA$, and negatively occurring, or F is of the form $\exists VA$, and positively occurring, then nothing changes.*

Definition 2.4.6 *Let $\Gamma \vdash \Delta$ be a sequent. Let I be an interpretation of $\Gamma \vdash \Delta$. The* unrestricted Skolem relations *$P_1, \ldots, P_s$ are obtained as follows:*

- *For every F of the form $\forall W_iA$ that positively occurs in $\Gamma \vdash \Delta$,*
    - *If F is true in $(D, [\,](d_{i,1}, \ldots, d_{i,l_i})$, then for all d,*

$$P_i(d_{i,1}, \ldots, d_{i,l_i}, d).$$

- *If F is false in $(D, [\,](d_{i,1}, \ldots, d_{i,l_i})$, then*

$$P_i(d_{i,1}, \ldots, d_{i,l_i}, d),$$

for all $d \in D$, such that A is false in $(D, [\,](d_{i,1}, \ldots, d_{i,l_i}; d))$.

- *Similarly for every F of the form $\exists W_i A$ that occurs negatively in $\Gamma \vdash \Delta$,*
  - *If F is true in $(D, [\,](d_{i,1}, \ldots, d_{i,l_i})$, then*

$$P_i(d_{i,1}, \ldots, d_{i,l_i}, d),$$

for all $d \in D$, such that A is true in $(D, [\,](d_{i,1}, \ldots, d_{i,l_i}; d)$.

  - *If F is false in $(D, [\,](d_{i,1}, \ldots, d_{i,l_i}; d)$, then, for all $d \in D$,*

$$P_i(d_{i,1}, \ldots, d_{i,l_i}, d).$$

Lemma 2.4.7

$$\Gamma \vdash \Delta \textit{ is true in } I \textit{ iff}$$

$$\Gamma \vdash \Delta \textit{ is true in } I \textit{ with } P_1, \ldots, P_s,$$

if $P_1, \ldots, P_s$ are the unrestricted Skolem relations based on the sequent $\Gamma \vdash \Delta$ and the interpretation I.

The proof can be obtained by induction following Definitions 2.2.1 and 2.4.5.

Lemma 2.4.8 *Let $\Gamma \vdash \Delta$ be a sequent. Let $W_1, \ldots, W_s$ be the variables to be Skolemised. Let $I = (D, [\,])$ be an interpretation of $\Gamma \vdash \Delta$. Let $P_1, \ldots, P_s$ be a set of Skolem relations.*
A restriction *of D and the P_i is a domain $\overline{D}$, and Skolem relations $\overline{P}_i$ satisfying the following:*

1. *$\overline{D} \subseteq D$,*
2. *if $d_1, \ldots, d_{l_i} \in \overline{D}$, then*

$$\overline{P}_i(d_1, \ldots, d_{l_i}, x) \Rightarrow P_i(d_1, \ldots, d_{l_i}, x).$$

The following is the case:

$$\Gamma \vdash \Delta \textit{ is true in } I \textit{ with } P_1, \ldots, P_s \textit{ iff}$$

$$\Gamma \vdash \Delta \textit{ is true in every restriction } \overline{I} = (\overline{D}, [\,]) \textit{ with } \overline{P}_1, \ldots, \overline{P}_s.$$

The proof can be obtained by induction on the structure of the formulae.
We can now combine 2.4.6 and 2.4.8 into:

Theorem 2.4.9 *Let $\Gamma \vdash \Delta$ be a sequent, Then: $\Gamma \vdash \Delta$ is true in every interpretation I iff for every interpretation $I = (D, [\,])$, for every sequence of Skolem relations $P_1, \ldots, P_s$, $\Gamma \vdash \Delta$ is true in I with $P_1, \ldots, P_s$.*

This is because every sequence of Skolem relations $P_1, \ldots, P_s$ is a restriction of the unrestricted Skolem relations.

Definition 2.4.10 *Let $I = (D, [\,])$ be an interpretation of $\Gamma \vdash \Delta$, let $P_1, \ldots, P_s$ be a set of Skolem relations. $P_1, \ldots, P_s$ are called* functional *if*

- *Whenever for $d_1, \ldots, d_{l_i} \in D$, both $P_i(d_1, \ldots, d_{l_i}, x)$ and $P_i(d_1, \ldots, d_{l_i}, y)$ hold, then $x = y$.*

Now the following is the case:

Lemma 2.4.11 *Let $\Gamma \vdash \Delta$ be a sequent, then $\Gamma \vdash \Delta$ is true in every interpretation $I = (D, [\,])$ with functional $P_1, \ldots, P_s$, iff the skolemisation $\Gamma' \vdash \Delta'$ is true in every interpretation of $\Gamma' \vdash \Delta'$.*

This is obtained by interpreting the Skolem functions as the functional Skolem relations.
We are almost finished now. It remains to show the following:

Lemma 2.4.12 *Let $D = (I, [\,])$ be an interpretation with Skolem relations $P_1, \ldots, P_s$. There exists a restriction $\overline{D}$, with functional $\overline{P}_1, \ldots, \overline{P}_s$.*

This completes the proof of Theorem 2.4.3.

2.5 Theorem of Herbrand

In this section we state what is usually called Herbrand's theorem. A sequent holds iff and only iff it possible to replace all formulae in the sequent by a finite number of instances, such that the resulting sequent holds.
According to ([Gal86]), this is not the orginal theorem of Herbrand, because the original theorem was stated in terms of *provability* instead of validity. However we will stick to the tradition and prove Herbrand's theorem, which is not from Herbrand:
First we define what an instance is:

Definition 2.5.1 *Let F be a formula. An* instance *of F is obtained by making substitutions for all quantifiers in F. So all $QX\,A$ in F are replaced by an $A[X := t]$.*

For example $p(a, b) \vee q(a)$ is an instance of $\forall X(\forall Y p(X, Y)) \vee q(X))$. Now we can state the theorem:

Theorem 2.5.2 *Let $\Gamma \vdash \Delta$ be a sequent with no negatively occurring $\exists$-quantifiers and no positively occurring $\forall$-quantifiers (typically obtained by Skolemisation). The following two are equivalent:*

1. *$\Gamma \vdash \Delta$ holds.*
2. *There is a $\Gamma' \vdash \Delta'$, which is obtained by replacing all formulae in $\Gamma \vdash \Delta$ by a finite (possibly 0) number of instances and this $\Gamma' \vdash \Delta'$ holds.*

Before we can give a proof we need some preparation.

Definition 2.5.3 *Let $\Gamma \vdash \Delta$ be a sequent with no negatively occurring $\exists$-quantifiers, and no positively occurring $\forall$-quantifiers. A* Herbrand interpretation *of $\Gamma \vdash \Delta$ is an interpretation $I = (D, [\ ])$, with*

- *D is the set of all terms that can be made from all variables, constants and function symbols in $\Gamma \vdash \Delta$.*
- *$[\ ]$ is defined in such a manner that $[t] = t$, for all terms. So, for all variables that occur in $\Gamma \vdash \Delta$,*

$$[X] = X.$$

For all constants c, that occur in $\Gamma \vdash \Delta$,

$$[c] = c,$$

For all functions f, that occur in $\Gamma \vdash \Delta$, (with arity n), $[f]$ is defined from:

$$[f](t_1, \ldots, t_n) = f(t_1, \ldots, t_n).$$

- *For the predicate symbols p in $\Gamma \vdash \Delta$, there are no conditions on $[p]$.*

All Herbrand interpretations have the same domain. They only differ in the interpretations of the predicate symbols. As a consequence a Herbrand interpretation is completely determined by the interpretations of the predicate symbols and it is possible to describe a Herbrand interpretation by summing up the literals which are true in it:

Definition 2.5.4 *Let $\Gamma \vdash \Delta$ be a sequent. The* Herbrand universe *of $\Gamma \vdash \Delta$ is the set of atoms that can be constructed from the predicate symbols, the function and constant symbols and the variables in $\Gamma \vdash \Delta$.*

Let $I = (D, [\ ])$ be a Herbrand interpretation of a sequent $\Gamma \vdash \Delta$. We define $\mathrm{HB}(I) =$

$$\{p(t_1, \ldots, t_n) \mid p \textit{ occurs in } \Gamma \vdash \Delta \textit{ with arity } n, \textit{ and } (t_1, \ldots, t_n) \in [p]\} \cup$$
$$\{\neg\, p(t_1, \ldots, t_n) \mid p \textit{ occurs in } \Gamma \vdash \Delta \textit{ with arity } n, \textit{ and } (t_1, \ldots, t_n) \notin [p]\}.$$

A Herbrand interpretation is completely determined by the set $\mathrm{HB}(I)$.

Lemma 2.5.5 *Let $\Gamma \vdash \Delta$ be a sequent with no negatively occurring $\exists$-quantifiers, and no positively occurring $\forall$-quantifiers. Let I be a Herbrand interpretation of $\Gamma \vdash \Delta$. Then $\Gamma \vdash \Delta$ is true in I if there is a $\Gamma' \vdash \Delta'$ obtained by replacing each formula in Γ' or Δ', by* one *instance which holds in I.*

Lemma 2.5.6 *Let $\Gamma \vdash \Delta$ be a sequent with no negatively occurring $\exists$-quantifiers, and no positively occurring $\forall$-quantifiers. $\Gamma \vdash \Delta$ holds in an interpretation iff $\Gamma \vdash \Delta$ holds in a Herbrand interpretation.*

2.6 Unification

In Definition 2.3.3 substitution was defined. Here we will look for minimal substitutions which make two terms equal.

If A is an atom then $A[X := t]$ is the result of replacing all free occurrences of X by t. In this notation the substitution is not considered as an independent object. The notation $A[X := t]$ is just a nice notation for a function $S(A, X, t)$. In this section, it is necessary to treat the substitution as an independent object, because, in order to define the minimal substitution, it is necessary to quantify over substitutions.
In the rest of this section we will not distinguish atoms and terms, because they have the same syntactical structure, and they behave the same under substitution. We will speak about terms, but we also mean atoms.

Definition 2.6.1 *A substitution is a finite set of assignments of the following form:*

$$\Theta = \{V_1 := t_1, \ldots, V_n := t_n\}.$$

It is not allowed that there are $V_i := t_i$, with $V_i = t_i$, because these are redundant. It must be the case that for no different $V_i := t_i$, and $V_j := t_j$, we have $V_i = V_j$. The intended meaning of a substitution is that in a term every occurrence of V_i will be replaced by t_i. We write $A\Theta$ for the application of Θ on A.

Definition 2.6.2 *Let*

$$\Sigma = \{V_1 := t_1, \ldots, V_n := t_n\}, \text{ and}$$

$$\Theta = \{W_1 := u_1, \ldots, W_m := u_m\}$$

be substitutions. The composition of Σ and Θ is defined as the set:

$$\Sigma \cdot \Theta = \{V := (V\Sigma)\Theta \mid V \neq (V\Sigma)\Theta\}.$$

The composition is always finite because it will contain not more than $n + m$ elements. It can be easily computed.

Lemma 2.6.3 *For every term A,*

$$A(\Sigma \cdot \Theta) = (A\Sigma)\Theta.$$

This is because it is true for every variable variable, by definition.
Composition of substitution is associative:

Lemma 2.6.4 *For all substitutions $\Sigma, \Theta,$ and Ξ,*

$$(\Sigma \cdot \Theta) \cdot \Xi = \Sigma \cdot (\Theta \cdot \Xi).$$

It is easily checked for each variable in Σ, Θ, or Ξ that this is true.
Now we define the most general unifier:

Definition 2.6.5 *A substitution Θ which makes two terms equal is called a* unifier. *Let A and B be two terms. A unifier Θ of A and B is called a* most general unifier *of A and B if for every unifier Σ, there is a substitution Ξ, such that $\Sigma = \Theta \cdot \Xi$.*

Let $A_1 = B_1, \ldots, A_n = B_n$ be a list of equations. A unifier of $A_1 = B_1, \ldots, A_n = B_n$ is a substitution Θ, for which $A_1\Theta = B_1\Theta, \ldots, A_n\Theta = B_n\Theta$. Θ is a most general unifier if for every unifier Σ of $A_1 = B_1, \ldots, A_n = B_n$, there is a Ξ, such that Σ can be written as $\Sigma = \Theta \cdot \Xi$. The term 'most general unifier' is often abbreviated as mgu.

We will also call the result $A\Theta$ an mgu, when Θ is an mgu of A and B.

We will call a substitution Θ a renaming substitution *if*

1. *For every variable V, the result $V\Theta$ is a variable,*
2. *For no two different variables V_1 and V_2 is it possible that $V_1\Theta = V_2\Theta$.*

It can be proven that the mgu is unique, up to renaming.
We will now prove the following important fact, due to ([Rob65]).

Theorem 2.6.6 *Let $A_1 = B_1, \ldots, A_n = B_n$ be a set of equations. The following procedure determines whether or not the set of equations has an mgu. The procedure also determines the mgu in the case that it exists. The algorithm maintains a state vector $(A_1 = B_1, \ldots, A_n = B_n, \Theta)$, where $A_1 = B_1, \ldots, A_n = B_n$ is a list of equations, and Θ is a substitution.*

Begin Conditions

Suppose that one wants to compute the mgu of the list of equations $A_1 = B_1, \ldots, A_n = B_n$. Begin with the state vector $(A_1 = B_1, \ldots, A_n = B_n, \emptyset)$.

Replacement Conditions:

Write the state vector as $(A_1 = B_1, \ldots, A_n = B_n, \Theta)$. Then:

1. *If one of the equations in the state vector has form $V = V$, for a variable V, then this equation can be deleted without affecting the other equations or Θ.*
2. *Any equation of the form $f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)$ can be replaced by the equations $t_1 = u_1, \ldots, t_n = u_n$, without affecting Θ. Note that if $n = 0$, then the equation is replaced by nothing.*
3. *An equation of the form $V = t$, or $t = V$, where V does not occur in t and $V \neq t$, can be removed, but all other equations $A_i = B_i$ have to be replaced by $A_i\{V := t\} = B_i\{V := t\}$, and Θ has to be replaced by $\Theta \cdot \{V := t\}$.*

End Conditions:

Write the state vector as $(A_1 = B_1, \ldots, A_n = B_n, \Theta)$. Then:

1. *If one of the equations has form* $f(t_1, \dots, t_n) = g(u_1, \dots, u_m)$*, where either* $f \neq g$*, or* $n \neq m$*, then report that the attempt to find an mgu has failed.*
2. *If one of the equations has form* $V = t$*, or* $t = V$*, where* V *does occur in* t*, but* $V \neq t$*, then report that the attempt to find an mgu has failed.*
3. *If there are no equations left, then the attempt to construct an mgu is successfull.* Θ *equals the mgu.*

We will prove that this algorithm is correct and terminating.
First we prove that it terminates. In order to do this we need the complexity of a term:

Definition 2.6.7 *The* complexity *of a term* t*, which will be written as* $\#t$*, is recursively defined as follows:*

1. *For a variable* V*,*
$$\#V = 1.$$
2. *For a term of the form* $f(t_1, \dots, t_n)$*, (possibly* $n = 0$*),*
$$\#f(t_1, \dots, t_n) = 1 + \#t_1 + \cdots + \#t_n.$$

The complexity of a list of equations $A_1 = B_1, \dots, A_n = B_n$ *equals* $\#A_1 + \#B_1 + \cdots + \#A_n + \#B_n$.

In order to see that the algorithm terminates, note that every replacement condition does one of the following two things:

1. It decreases the number of variables in the equations.
2. It does not change the number of variables in the equations, but it reduces the total complexity of the equations.

(In case 3 the total number of variables decreases, because V does not occur in t. Checking the other replacement conditions is trivial). Because of this only a finite number of replacements will be made in the equations and the algorithm ends.
In order to prove the correctness of the algoritm we give the invariant of the algorithm:

INV1 Let (E, Θ), be a state vector that is constructed by the algorithm of Theorem 2.6.6. If the initial set of equations has a unifier, then whenever Σ is an mgu of E, the composition $\Theta \cdot \Sigma$ is an mgu of the initial set of equations. If Ξ is a unifier of the initial set of equations, then Ξ can be written as $\Theta \cdot \Sigma$.

INV2 If the initial set of equations does not have a unifier, then the algorithm will not enter a state (E, Θ), in which E has a unifier.

INV1 will ensure that the algorithm will give an mgu if one exists. INV2 ensures that if the algorithm gives an mgu, that then there exists a unifier of the initial set of equations.
We will prove the correctness of the procedure by proving that the replacements steps preserve the invariants. First some preparation:

Lemma 2.6.8 *For a single equation $V = t$, the following holds:*

1. *If $V \neq t$, and V occurs in t, then for no substitution Σ it is possible that $V\Sigma = t\Sigma$.*

2. *If $V = t$, the equation $V = t$ has mgu $\emptyset$.*

3. *If $V \neq t$, and V does not occur in t, then $V = t$ has mgu $\{V := t\}$.*

Proof

1. Because V occurs strictly in t, it will be the case that $V\Sigma$ occurs strictly in $t\Sigma$. No term can be equal to a strict subterm of itself.

2. If $V = t$, then obviously $V\emptyset = t\emptyset$. Because every substitution can be written as $\Sigma = \emptyset \cdot \Sigma$, the empty substitution is the mgu.

3. $V\{V := t\} = t$, and $t\{V := t\} = t$, because V does not occur in t. Hence $\{V := t\}$ is a unifier. Now assume that $V\Sigma = t\Sigma$. Then $\Sigma = \{V := t\} \cdot \Sigma$. This is because of the following: If $V \neq W$, then $W\{V := t\} = W$, and so $W\Sigma = W(\{V := t\} \cdot \Sigma)$. For V holds $V(\{V := t\} \cdot \Sigma) = t\Sigma = V\Sigma$.

Lemma 2.6.9 *The equation $f(t_1, \ldots, t_n) = g(u_1, \ldots, u_m)$ with $f \neq g$, or $n \neq m$, has no unifier.*

Proof
Let Σ be a substitution. Then:

$$f(t_1, \ldots, t_n)\Sigma = f(t_1\Sigma, \ldots, t_n\Sigma), \text{ and}$$

$$g(u_1, \ldots, u_m)\Sigma = g(u_1\Sigma, \ldots, u_m\Sigma).$$

Still $f \neq g$, or $n \neq m$, and the terms are not equal...

Lemma 2.6.10 *Substitution Σ is a unifier of*

$$f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n) \textit{ iff}$$

Σ is a unifier of

$$t_1 = u_1, \ldots, t_n = u_n.$$

Proof
Because

$$f(t_1, \ldots, t_n)\Sigma = f(t_1\Sigma, \ldots, t_n\Sigma),$$
$$f(u_1, \ldots, u_n)\Sigma = f(u_1\Sigma, \ldots, u_n\Sigma), \text{ and}$$
$$f(t_1\Sigma, \ldots, t_n\Sigma) = f(u_1\Sigma, \ldots, u_n\Sigma) \text{ iff}$$
$$t_1\Sigma = u_1\Sigma, \ldots, t_n\Sigma = u_n\Sigma.$$

Now we are ready to prove the correctness of the algorithm: We first prove that INV1 is preserved. During the proof the state vector is written as (E, Θ).

- We show that the invariant is produced by the initial state. Initially E is the initial set of equations, and $\Theta = \emptyset$. If the the initial set E has a unifier, and Θ is an mgu of E, then obviously $\emptyset \cdot \Theta$ equals the mgu.

- Let (E', Θ) be obtained from (E, Θ) by the first or second replacement rule. We prove that $\text{INV1}(E, \Theta) \Rightarrow \text{INV1}(E', \Theta)$. In both cases we have Σ is unifier of E' iff Σ is a unifier of E. (In the second case by Lemma 2.6.10). As a consequence E and E' have the same set of mgu's.

- Let $(E\{V := t\}, \Theta \cdot \{V := t\})$ be obtained from $(E \cup \{V = t\}, \Theta)$ by an application of the third replacement rule. We prove $\text{INV1}(E\{V := t\}, \Theta \cdot \{V := t\})$ from $\text{INV1}(E \cup \{V = t\}, \Theta)$. Assume that the initial set of equations has a unifier. Let Σ be an mgu of $E\{V := t\}$. We have to show that

  - $(\Theta \cdot \{V := t\}) \cdot \Sigma$ is an mgu of the initial set of equations.

  By associativity of $\cdot$,

  $$(\Theta \cdot \{V := t\}) \cdot \Sigma = \Theta \cdot (\{V := t\} \cdot \Sigma).$$

  We are finished if we manage to prove that $\{V := t\} \cdot \Sigma$ is an mgu of $E \cup \{V = t\}$, because then we can use $\text{INV1}(E \cup \{V = t\}, \Theta)$.
  By Lemma 2.6.8, $\{V := t\}$ is an mgu of $V = t$. Then
  $E \cup \{V = t\}(\{V := t\} \cdot \Sigma) = (E\{V := t\} \cup \{t = t\})\Sigma$. As a consequence $\{V := t\} \cdot \Sigma$ unifies $E \cup \{V = t\}$.
  Now assume that Ξ is a unifier of $E \cup \{V = t\}$. We want to write $\Xi = (\{V := t\} \cdot \Sigma) \cdot \Xi'$. Because Ξ must unify $V = t$, it must be the case that Ξ can be written as $\Xi = \{V := t\} \cdot \Phi$. Φ must be a unifier of $E\{V := t\}$, and hance Φ can be written as $\Sigma \cdot \Xi'$. We are finished now: Write

  $$\Xi = \{V := t\} \cdot (\Sigma \cdot \Xi') = (\{V := t\} \cdot \Sigma) \cdot \Xi'.$$

- In the first and second end condition. There is an $A = B \in E$, that has no unifier. Assume that the initial set of equations has a unifier. Then it must be possible to write the mgu as $\Theta \cdot \Sigma$. This is impossible because E has no unifier. Therefore the initial set of equations cannot have a unifier.

- It remains to show that in the third end conditions, Θ is the mgu.

We now prove that if the initial set of equations has no unifier, the algorithm will not enter a state (E, Θ), where E has a unifier.

- If the initial set of equations does not have a unifier, then in the initial state (E, Θ), obviously E does not have a unifier.
- If (E', Θ) is obtained from (E, Θ) by the first or second replacement rule, then E has a unifier iff E' has a unifier. Therefore E' has no unifier.
- Let $(E\{V := t\}, \Theta \cdot \{V := t\})$ be obtained from $(E \cup \{V = t\}, \Theta)$. If $E\{V := t\}$ has a unifier Σ, then certainly $\{V := t\} \cdot \Sigma$ is a unifier of $E \cup \{V = t\}$, by Lemma 2.6.8.

We have now proven the correctness of the algorithm of Theorem 2.6.6.

Sometimes it is better to use another algorithm. The following algorithm is better suited for proving properties of unification than the algorithm of Theorem 2.6.6. It is not so efficient, but that is not what it is designed for.

Theorem 2.6.11 *Let A and B be two terms, that have an mgu. The following procedure will obtain an mgu. The mgu will be obtained as a concatenation of the following form: $\Theta = \Sigma_1 \cdot \ldots \cdot \Sigma_n$. The procedure begins with $\Theta_0 = \{\}$. When Θ_i is defined as $\Sigma_1 \cdot \ldots \cdot \Sigma_i$, the procedure will construct another Σ_{i+1} as long as $A\Theta_i$ and $B\Theta_i$ are not equal.*

This is done as follows:

- *Put $T_1 = A\Theta_i$, and $T_2 = B\Theta_i$. Then as long as long as neither T_1, nor T_2 is a variable do:*
  - *If $T_1 = f(t_1, \ldots, t_n)$ $T_2 = f(u_1, \ldots, u_n)$, and $T_1 \neq T_2$, then one pair t_i, u_i must be unequal. Replace T_1 by t_i, and replace T_2 by u_i.*

  Now, either T_1 or T_2 is a variable.

  1. *If T_1 is a variable, then put $\Sigma_i = \{T_1 := T_2\}$.*
  2. *If T_2 is a variable, then put $\Sigma_i = \{T_2 := T_1\}$.*

The Σ_i are called mesh substituents*.*

A computation of this algorithm can be transformed into a computation of the algorithm in Theorem 2.6.6. It is not necessary to include a test because the procedure assumes that there exists an mgu.

2.7 Resolution Theorem Proving

In this section we define resolution theorem proving. We begin by defining the normal form that is used, and then we will introduce the resolution rule. After that we prove the soundness and completeness of resolution.

Definition 2.7.1 *A* clause *is a finite set of literals. The* meaning *of* $\{A_1, \ldots, A_p\}$ *is* $\forall \overline{X}(A_1 \vee \cdots \vee A_p)$*, where* $\overline{X}$ *are the variables that occur in the* A_i*. The meaning of* $\emptyset$ *is* $\bot$*, where* $\bot$ *is a special formula that cannot be true in an interpretation. A clause is* ground *if there are no variables in it. The effect of a substitution on a clause is defined memberwise.*

For example, the meaning of $\{p(X), q(X, Y)\}$ is $\forall X \forall Y(p(X) \vee q(X, Y))$. The meaning of $\{P, R(0)\}$ is $P \vee R(0)$.

Theorem 2.7.2 *Let* $\Gamma \vdash \Delta$ *be a sequent. There exist algorithms which transform* $\Gamma \vdash \Delta$ *into a finite set of clauses* $\{c_1, \ldots, c_n\}$*, such that* $\Gamma \vdash \Delta$ *holds if and only if* $\{c_1, \ldots, c_n\}$ *is unsatisfiable.*

Proof
The proof proceeds in a number of steps.

1. If $\Gamma \vdash \Delta$ contains free variables, then add quantors: If a formula F, with free variable V occurs in Γ, then F is replaced by $\forall V F$. If a formula F, with free variable V occurs in Δ, then F is replaced by $\exists V F$. It is easily checked that $\Gamma \vdash \Delta$ holds iff the result of these replacements holds.

2. After that $\Gamma \vdash \Delta$ can be Skolemised to obtain a sequent $\Gamma' \vdash \Delta'$. By Theorem 2.4.3,
$$\Gamma \vdash \Delta \text{ holds iff } \Gamma' \vdash \Delta' \text{ holds.}$$
There exists an algorithm which constructs the Skolemisation.

3. After this we have a sequent $\Gamma \vdash \Delta$ with no positively occurring $\exists$-quantifiers, and no negatively occurring $\forall$-quantifiers. Write $\Gamma' \vdash \Delta'$ as $\{A_1, \ldots, A_p\} \vdash \{B_1, \ldots, B_q\}$, and replace it by
$\{A_1, \ldots, A_p, \neg B_1, \ldots, \neg B_q\} \vdash$. Replace variables in such a manner that no different quantifiers have the same variable. Then apply the following rewrite rules as long as possible:

$$\begin{array}{lll}
A \rightarrow B & \Rightarrow & \neg A \vee B, \\
\neg (A \rightarrow B) & \Rightarrow & A \wedge \neg B, \\
\neg (A \wedge B) & \Rightarrow & \neg A \vee \neg B, \\
\neg (A \vee B) & \Rightarrow & \neg A \wedge \neg B.
\end{array}$$

$$\begin{array}{lll}
(\forall X P) \wedge Q & \Rightarrow & \forall X(P \wedge Q), \\
P \wedge \forall X Q & \Rightarrow & \forall X(P \wedge Q), \\
(\forall X P) \vee Q & \Rightarrow & \forall X(P \vee Q), \\
P \vee \forall X Q & \Rightarrow & \forall X(P \vee Q).
\end{array}$$

$$\begin{array}{lll}
(A \wedge B) \vee C & \Rightarrow & (A \vee C) \wedge (B \vee C), \\
A \vee (B \wedge C) & \Rightarrow & (A \vee B) \wedge (B \wedge C).
\end{array}$$

4. After that the following rules are applied:

   $$\forall X(P \wedge Q) \quad \Rightarrow \quad \forall X P \wedge \forall X Q,$$

   As long as the sequent can be written as $\Gamma, A \wedge B \vdash$, this is replaced by $\Gamma, A, B \vdash$.

5. The result of the previous step will be of the form: $\{c_1, \ldots, c_n\} \vdash$, where each c_i is of the form

   $$c_i = \forall X_{i,1} \cdots \forall X_{i,v_i}(L_1 \vee \cdots \vee L_{i,l_i}).$$

   Then the sequent $\{c_1, \ldots, c_m\} \vdash$ can be replaced by the clause set $\{\{L_{1,1}, \ldots, L_{1,l_1}\}, \ldots, \{L_{n,1}, \ldots, L$

The algorithm given here has an exponential worst case complexity, which is caused by the factorisation rules:

$$(A \wedge B) \vee C \Rightarrow (A \vee C) \wedge (B \vee C), \text{ and } A \vee (B \wedge C) \Rightarrow (A \vee B) \wedge (B \vee C).$$

However in practice this algorithm behaves quite well because the formulae which would cause exponential growth, do not occur in real life problems.
If one however finds this unacceptable it is possible to introduce labels in the translation. (See [Min88]).
It has turned out that making this translation is a rather subtle art. Small variations in Skolemisation, or in the manner of introducing the labels may cause hyperexponential differences in the resolution proof that can be found.
We have now reduced the problem of proving a sequent $\Gamma \vdash \Delta$ to the problem of proving that a certain clause set $\{c_1, \ldots, c_n\}$ is unsatisfiable. This can be done by resolution.

Definition 2.7.3 *We define resolution and factoring:*

Resolution *Let*

$$c_1 = \{A_1, \ldots, A_p\}, \text{ and}$$
$$c_2 = \{B_1, \ldots, B_q\}.$$

be two clauses with no overlapping variables, such that A_1 and $\neg\, B_1$ are unifiable, with most general unifier Θ. Then the clause

$$\{A_2\Theta, \ldots, A_p\Theta, B_2\Theta, \ldots, B_q\Theta\}$$

is a resolvent *of c_1 and c_2. The literals A_1 and $\neg\, B_1$ are called the literals* resolved upon.

Factorisation *Let*

$$c = \{A_1, \ldots, A_p\}$$

be a clause, such that A_1 is unifiable with one of the A_i, with $2 \leq i < p$. Let Θ be the most general unifier. Then the clause

$$\{A_2\Theta, \ldots, A_{i-1}, A_{i+1}, \ldots, A_p\Theta\}$$

is a factor *of c. The literals A_1 and A_2 are called the literals* factored *upon.*

These rules will be used as follows. One starts with the initial clause set and tries to derive the empty clause by exhaustive search. It is the case that the empty clause will be derived if and only if the initial clause set is unsatisfiable. We will now give an example, then warn for some subtleties in the definition of the resolution rule, and then prove that resolution is sound and complete.

Example 2.7.4 *$\{p(0), q(0)\}$ is a resolvent of $\{p(X), q(X), r(X)\}$ and $\{\neg\, r(0)\}$.*
$\{p(X, X, Y), q(X, X, Y)\}$ and $\{\neg\, p(X, Y, Y), r(X, Y, Y)\}$ resolve to
$\{q(X, X, X), r(X, X, X)\}$.
The clause $\{p(X, X)\}$ is a factor $\{p(X, Y), p(Y, X)\}$.

Some care has to be taken in the application of the resolution rule. (See [Lei88]). In the definition above, the construction of a resolvent goes as follows:

1. A_1 and B_1 are deleted resulting in $\{A_2, \ldots, A_p\}$ and $\{B_2, \ldots, B_q\}$.
2. After that Θ is applied to obtain $\{A_2\Theta, \ldots, A_p\Theta\}$ and $\{B_2, \ldots, B_q\}$. These sets are joined.

One might be tempted to do it in different order:

1. First replace $\{A_1, \ldots, A_p\}$ by $\{A_1\Theta, \ldots, A_p\Theta\}$, and $\{B_1, \ldots, B_q\}$ by $\{B_1\Theta, \ldots, B_q\Theta\}$.
2. Then delete $A_1\Theta$ from $\{A_1\Theta, \ldots, A_p\Theta\}$, and $B_1\Theta$ from $\{B_1\Theta, \ldots, B_q\Theta\}$ and join the results.

This, however, is a different resolution rule, because it may be the case that $A_1\Theta = A_i\Theta$, for an $i \neq 1$, or $B_1\Theta = B_j\Theta$, for a $j \neq 1$. In the first case, in the second version of the resolution rule $A_i\Theta$ will be not present in the resolvent, whereas in the first version $A_i\Theta$ would occur in the resolvent.
When, for example $\{p(X, Y), p(Y, X)\}$ and $\{\neg\, p(0, 0)\}$ resolve according to the first version, the result will be $\{p(0, 0)\}$. When they resolve according to the second version, the result will be $\{\}$.
The first version of resolution is weaker than the second version, because the clause, derived with the second version is a subset of the clause, derived with the first version.
We will now prove the soundness of resolution by proving the following: If the meaning of a clause c is true in an interpretation, and c' is a factor of c, then the meaning of c' is true in the interpretation. Here we use the stronger notion of resolution. If the meanings of c_1 and c_2 are true in an interpretation, then the meaning of every possible resolvent of c_1 and c_2 is true in the interpretation.

Lemma 2.7.5 *Let A be a literal. Let $A[X := t]$ be an instance of A, such that X does not occur in t. Let $I_1 = (D, [\ ]_1)$ be an interpretation of A, and let $I_2 = (D, [\ ]_2)$ be an interpretation of $A[X := t]$, such that*

1. *For all function and constant symbols f and c,*

$$[f]_1 = [f]_2 \text{ and } [c]_1 = [c]_2.$$

2. *For all variables V in A, except X,*

$$[V]_1 = [V]_2.$$

3.

$$[X]_1 = [t]_2.$$

Then $(D, [\ ]_1)$ makes A true iff $(D, [\ ]_2)$ makes $A[X := t]$ true.

The proof is obtained by induction on the structure of A.

Lemma 2.7.6 *Resolution and factoring are sound rules. We prove the following:*

1. *Let c be a clause, let c' be a factor of c. Let I be an interpretation of the meaning of c, such that the meaning of c is true in I. Then the meaning of c' is true in I.*

2. *Let c_1 and c_2 be clauses. Let c be a resolvent of c_1 and c_2. If I is an interpretation of the meanings of c_1, and c_2, such that both are true in I, then the meaning of c is true in I.*

Proof
First we prove

- Let $c\Theta$ be an instance of a clause c, such that for no variable V,

  $$V \text{ occurs in } V\Theta \text{ and } V \neq V\Theta.$$

  If the meaning of c is true in I, then the meaning of $c\Theta$ is true in I.

It is sufficient to prove this for simple Θ of the form $\Theta = \{X := t\}$. More complex substitutions can be obtained by iteration. Write the variables that occur in t as $Z_1, \ldots, Z_m$. Write the meanings of c and $c\Theta$ as

$$\forall X Y_1 \cdots Y_n (A_1 \vee \cdots \vee A_p), \text{ and}$$

$$\forall Z_1 \cdots Z_m Y_1 \cdots Y_n (A_1[X := t] \vee \cdots \vee A_p[X := t]).$$

(It is not excluded that some of the Z_i and Y_j are equal) Assume that the meaning of c is true in an interpretation I. This implies that in every interpretation $I_1 = (D, [\ ]^{X}_{x}\,{}^{Y_1}_{y_1} \cdots {}^{Y_n}_{y_n})$, the formula

$$A_1 \vee \cdots \vee A_p$$

is true.
We will prove from this that $A_1[X := t] \vee \cdots \vee A_p[X := t]$ is true in every interpretation $I_2 = (D, []^{Z_1}_{z_1} \cdots {}^{Z_m}_{z_m}\,{}^{Y_1}_{y_1} \cdots {}^{Y_n}_{y_n})$.
From an arbitrary I_2, an interpretation I_1 can be constructed by putting $[X]_1 =$

$[t]_2$. Then by Lemma 2.7.5, A_i is true in I_1 iff $A_i[X := t]$ is true in I_2. It follows that one of the $A_i[X := t]$ must be true in I_2, and hence the meaning of $c\{X := t\}$ must be true in I.

From what we have now the soundness of factorisation follows. We prove the soundness of the strong version of resolution. The soundness of the weak version follows immediately from it.

Let $c_1 = \{A_1, \ldots, A_p\}$ and $c_2 = \{B_1, \ldots, B_q\}$ with $A_1 = \neg\, B_1$. Let $I = (D, [\,])$ be an interpretation of c_1 and c_2. Assume that both the meanings of c_1 and c_2 are true in I. Let $I' = (D, [\,]^{V_1}_{v_1} \cdots {}^{V_n}_{v_n})$ be an interpretation of $A_2 \vee \cdots \vee A_p \vee B_2 \vee \cdots \vee B_q$. I' is also an interpretation of $A_1 \vee \cdots \vee A_p$ and $B_1 \vee \cdots \vee B_q$. Since by assumption both must be true in I', and it is impossible that both B_1 and A_1 are true in I', either $A_2 \vee \cdots \vee A_p$, or $B_2 \vee \cdots \vee B_q$ must be true in I'. In both cases $A_2 \vee \cdots A_p \vee B_2 \vee \cdots \vee B_q$ is true in I'. Because of this the meaning of c is true in I.

We have now proven the soundness of resolution. It remains to prove the completeness. For this we prove first that every unsatisfiable set of ground clauses has a resolution refutation. After that we prove the so called *lifting lemma*.

Lemma 2.7.7 *Let $C = \{c_1, \ldots, c_n\}$ be a finite set of propositional clauses. If C is unsatisfiable then C has a resolution refutation.*

Proof

Let C_1 and C_2 be two finite clause sets. We write $C_1 \sqsubset C_2$ iff C_1 can be obtained from C_2 by deleting some literals from some clauses. This relation can be used for induction because there is only a finite number of literals to delete. If $C_1 \sqsubset C_2$, and C_2 is unsatisfiable, then C_1 is unsatisfiable. We prove Lemma 2.7.7 by induction on this relation.

Basis If for all $c_i \in C$, the length is at most one, then either C contains the empty clause, or a complementary pair $\{A\}$ and $\{\neg\, A\}$. In both case $\emptyset$ can be derived rather quickly.

Step If for some $c_i \in C$, the length is at least two, then let A be a literal in c_i. Define

- $C\backslash A$ is the result of deleting A from all clauses in C, in which A occurs.
- $C \to A$ is the result of replacing all clauses in C, with length at least two and in wich A occurs by the clause $\{A\}$.

Both $C\backslash A$ and $C \to A$ are unsatisfiable. By the induction hypothesis both have a derivation of $\emptyset$. Consider the derivation of $\emptyset$ from $C\backslash A$. There are two possibilities:

1. None of the clauses in C that contained A has been used. In that case the empty clause can also be derived from C.

2. One of the clauses that contained A in C has been used. Then the derivation of $\emptyset$ from $C\backslash A$ can be replaced by a derivation of $\{A\}$ from C. Because $\emptyset$ can be derived from $C \to A$, the empty clause can be derived from $C \cup \{\{A\}\}$, and hence from C.

Now we have proven the propositional completeness. We will now prepare for the proof of the Lifting Lemma.

Lemma 2.7.8 *Let c be a clause with an instance $c\Theta$. Let A be a literal in $c\Theta$. If there is more than one literal $B \in c$, for which $A = B\Theta$, then c has a (possibly iterated) factor $c\Sigma$, such that there is only one literal in $c\Sigma$ of which A is an instance.*

Proof
We will reduce the number n of literals $B \in c$ for which $B\Theta = A$ by computing a factor $c\Sigma$, until $n = 1$. If $n > 1$, then there are at least two literals B_1 and B_2, for which $B_1\Theta = B_2\Theta = A$. Because B_1 and B_2 are unifiable, they have an mgu Σ. Then Θ can be written as $\Theta = \Sigma \cdot \Xi$. Then $c\Theta = c(\Sigma \cdot \Xi) = (c\Sigma)\Xi$. Because of this $c\Theta$ is an instance of $c\Sigma$. There are strictly less literals $B \in c\Sigma$ for which $B\Xi = A$, because B_1 and B_2 are unified in $c\Sigma$.

Lemma 2.7.9 *(Lifting Lemma) Let c_1 and c_2 be two clauses with instances $\overline{c}_1$ and $\overline{c}_2$. If $\overline{c}_1$ and $\overline{c}_2$ have a resolvent $\overline{c}$, then c_1 and c_2 have a resolvent c, such that $\overline{c}$ is an instance of c.*

Proof
Write

$$\overline{c}_1 = \{\overline{A}_1, \dots, \overline{A}_p\},$$
$$\overline{c}_2 = \{\overline{B}_1, \dots, \overline{B}_q\}.$$

where $\overline{A}_1 = \neg\, \overline{B}_1$, and $\overline{A}_1$ and $\overline{B}_1$ are the literals resolved upon.
By Lemma 2.7.8 there exist (possibly iterated) factors of c_1 and c_2, such that $\overline{A}_1$ and $\overline{B}_1$ are an instance of only one literal in c_1 and c_2. We from now on assume that c_1 and c_2 are already factored. Then we have that $\overline{c}_1\backslash\{\overline{A}_1\}$ is an instance of $c_1\backslash\{A_1\}$, because no other literal than A_1 in c_1 will have $\overline{A}_1$ as instance. In the same way $\overline{c}_2\backslash\{\overline{B}_1\}$ is an instance of $c_2\backslash\{B_1\}$.
If we assume that c_1 and c_2 have no overlapping variables, then there exists one substitution Θ, such that

1. $A_1\Theta = \neg\, B_1\Theta$, and
2. $\overline{c} = \{A_2\Theta, \dots, A_p\Theta, B_2\Theta, \dots, B_q\Theta\}$.

Because of this A_1 and $\neg\, B_1$ are unifiable, and c_1 and c_2 have a resolvent. Define Σ as the mgu of A_1 and $\neg\, B_1$. By definition of mgu, Θ can be written as $\Theta = \Sigma \cdot \Xi$. The resolvent equals $\{A_2\Sigma, \dots, A_p\Sigma, B_2\Sigma, \dots, B_q\Sigma\}$, from which $\{\overline{A}_2, \dots, \overline{A}_p, \overline{B}_2, \dots, \overline{B}_q\}$ will be obtained by Ξ.

Theorem 2.7.10 *Resolution is complete. If a clause set C is unsatisfiable, then the empty clause can be derived from it.*

Proof
Assume that C is unsatifiable. Then:

1. By Herbrands theorem, (Theorem 2.5.2), there exists a finite set $\{c_1, \dots, c_n\}$ of ground instances of elements of C, such that $\{c_1, \dots, c_n\}$ is unsatisfiable.
2. By Lemma 2.7.7 this set has a resolution refutation.
3. By Lemma 2.7.9 this ground refutation can be lifted to a non-ground resolution refutation of C.

Example 2.7.11 *Suppose one wants to prove the following sequent by resolution:*

$$\forall X(p(X) \rightarrow q(Y)) \vdash \neg\ [\forall X(\neg\ p(X) \wedge \neg\ q(X))] \rightarrow \exists Y q(Y).$$

First an existential quantifier is added:

$$\exists Y \forall X(p(X) \rightarrow q(Y)) \vdash \neg\ [\forall X(\neg\ p(X) \wedge \neg\ q(X)) \rightarrow \exists Y q(Y).$$

Then the sequent can be Skolemised:

$$\forall X(p(X) \rightarrow q(c)) \vdash \neg\ (\neg\ p(d) \wedge \neg\ q(d)) \rightarrow \exists Y q(Y).$$

The right side is brought to the left:

$$\forall X(p(X) \rightarrow q(c)), \neg\ [(\neg\ (\neg\ p(d) \wedge \neg\ q(d)) \rightarrow \exists Y q(Y)] \vdash .$$

The rewrite rules give:

$$\forall X(\neg\ p(X) \vee q(c)), p(d) \vee q(d), \forall Y[\neg\ q(Y)] \vdash .$$

This results in the following set of clauses:

(1) $\{\neg\ p(X), q(c)\}$
(2) $\{p(d), q(d)\}$
(3) $\{\neg\ q(Y)\}$.

Then, there is the following resolution refutation: We give both the ground refutation, and the non-ground refutation:

(1)	$\{\neg\ p(d), q(c)\}$	$\{\neg p(X), q(c)\}$	*(initial clause)*
(2)	$\{p(d), q(d)\}$	$\{p(d), q(d)\}$	*(initial clause)*
(3)	$\{\neg\ q(d)\}$	$\{\neg\ q(Y)\}$	*(initial clause)*
(4)	$\{\neg\ q(c)\}$	$\{\neg\ q(Y)\}$	*(initial clause)*
(5)	$\{\neg\ p(d)\}$	$\{\neg\ p(Y)\}$	*(from 1 and 4)*
(6)	$\{q(d)\}$	$\{\ q(d)\ \}$	*(from 2 and 5)*
(7)	$\{\}$	$\{\}$	*(from 3 and 6)*

2.8 Refinements of Resolution

In this section we present the standard ordering refinements of resolution. (See [CL73], and [Lov78]).

Definition 2.8.1

- *An L-order $\prec$ is an order on literals with the following property:*

  $$A \preceq B \textit{ implies } A\Theta \preceq B\Theta, \textit{ for all substitutions } \Theta.$$

  This property is called liftability.

- *An A-order is an L-order with the following property:*

  $$A \prec B \textit{ implies } \pm A \prec \pm B.$$

- *An* interpretation *is a set of literals I, with the following properties:*
  - *If $A \in I$, then for all instances $A\Theta$ of A, $A\Theta \in I$.*
  - *If $A \notin I$, then for all instances $A\Theta$ of A, $A\Theta \notin I$.*
  - *For each atom A exactly one of A and $\neg A$ is in I.*

  A clause c is called true *in I if $I \cap c \neq \emptyset$. Otherwise it is called* false.

- *Let C a clause set. An* indexing *of C is obtained by replacing each clause $\{A_1, \ldots, A_p\}$ by $\{A_1{:}n_1, \ldots, A_p{:}n_p\}$. Here for each clause the n_i can be chosen arbitrarily.*

The notions of L-order and interpretation can be slightly generalised by considering only literals that are instances of literals in a certain clause set.
The common definition for liftability is:

$$A \prec B \text{ implies } A\Theta \prec B\Theta.$$

The liftability given above is slightly more general, because the substitution is allowed to unify literals. There exist orderings which satisfy the weaker condition, but which do not satisfy the stronger condition, which can be lifted: An example is: $A \prec B$ if $A \neq p(0)$, and $B = p(0)$.
Using the previous we define the following refinements of resolution:

Definition 2.8.2

- **L-ordered resolution, A-ordered resolution**.
  L-ordered resolution is obtained from ordinary resolution, by taking an L-order $\prec$, and adding the following conditions:
  1. *When a resolvent is constructed the literals resolved upon must be maximal.*

2. *When a factor is constructed, one of the literals factored upon must be maximal.*

A-ordered resolution is obtained in the same manner, but taking an A-order instead of an L-order.

- **Semantic Resolution**.
  Semantic resolution is obtained from ordinary resolution, by fixing an interpretation I, and adding the following conditions:

  1. *When a resolvent is constructed, one of the clauses resolved upon must be false.*
  2. *Only factors of false clauses can be constructed.*

- **Hyperresolution**.
  Hyperresolution is obtained by making leaps in semantic resolution. If a clause is true in I, then it can be written as: $\{A_1, \ldots, A_p, B_1, \ldots, B_q\}$, where $\{A_1, \ldots, A_p\} \subseteq I$, and $\{B_1, \ldots, B_q\} \cap I = \emptyset$. If this clause will be resolved with a false clause, the result will contain one true literal less. Then as long as the result contains true literals it will have to resolve with a false clause. This happens p times. After the p-th time the result is false. Hyperresolution is obtained from semantic resolution by only keeping the final false clauses. So, hyperresolution is obtained by imposing the following conditions:

  1. *A true clause, containing q true literals has to be resolved q times with a false clause. Only the results of these iterated resolution steps are kept.*
  2. *Factorisation is only allowed on false clauses.*

- **A-ordered hyperresolution, A-ordered semantic resolution**.
  A-ordered hyperresolution is obtained by combining the hyperresolution refinement with A-ordered resolution.

- **Lock Resolution**.
  Lock resolution is obtained as follows. The literals of the initial clauses are indexed. After that resolution and factoring are modified as follows:

  1. *Only literals with maximal indices can be resolved upon. When a resolvent is constructed, the literals in the result inherit the indices from the original clauses.*
  2. *When a factor is constructed one of the literals factored upon must have a maximal index.*

Then we have:

Theorem 2.8.3 *All refinements, mentioned in Definition 2.8.2 are complete.*

Proof
The proof consists of two parts.

1. First prove propositional completeness for all of the refinements.

2. Then prove that ground refutations in each refinement can be lifted.

We begin by proving propositional completeness. The completeness proof of L-ordered resolution can be obtained by making a small adaptation in the proof of Lemma 2.7.7. In the step the following will be changed: If one of the clauses has a length of at least two, then a minimal literal occurring in the clauses of length at least two will be chosen (as A). The rest of the proof can be the same because the clause $\{A\}$ can be derived.

The completeness proof of lock resolution can be obtained in the same manner: From the clauses of length at least two select a literal with minimal index.

We will prove the propositional completeness of the remaining refinements by proving the completeness of A-ordered semantic resolution. From this follows the completeness of hyperresolution, A-ordered hyperresolution, and semantic resolution. We have to be a bit careful because of the restriction in factoring on the non-ground level. In order to avoid problems in lifting we take all true clauses as multisets, and all false clauses as ordinary sets. Then the proof of Lemma 2.7.7 can be adapted as follows:

Basis If for all $c_i \in C$, the number of false literals in c_i equals at most 1, then every false clause will be a unit clause, or the empty clause.

Step If for some $c_i \in C$, there is more than one true literal in c_i, then choose A as a minimal true literal from the clauses that have more than one true literal. The rest of the proof is the same.

This results in an A-ordered semantic refutation which has no implicit factoring in true clauses.
In order to prove the possibility of lifting for all the refinements here it is sufficient to prove the following, for each refinement:
If $\{A_1, \ldots, A_p\}$ has instance $\{A_1\Theta, \ldots, A_p\Theta\}$, then

1. If $A_1\Theta$ is allowed to be resolved upon by the refinement, then if $A_1\Theta$ is an instance of more than one A_i, then one of the A_i is allowed to be factored upon by the refinement. Using this Lemma 2.7.8 can be proven.

2. If $A_1\Theta$ is allowed to be resolved upon by the refinement, and $A_1\Theta$ is an instance of only A_1, then A_1 is allowed to be resolved upon by the refinement.

For L-ordered resolution we note: If $A_1\Theta$ is an instance of $A_1, \ldots, A_n$, and $A_1\Theta$ is maximal in $c\Theta$, then one of the A_i is maximal in c. Suppose that none of the A_i is maximal. Then there is a B, such that $A_1 \prec B$, and $A_1\Theta \neq B\Theta$,

and $A_1\Theta \not\prec B\Theta$. This is a violation of the liftability property, as defined in Definition 2.8.1.
For lock resolution we note that if $A_1{:}\,\Theta$ is maximal in $c\Theta$, then all $A_i{:}n$ which have $A_1{:}\,\Theta$ as instance have the same index. As a consequence these are also maximal.
For A-ordered semantic resolution we note the following: If $A_1\Theta$ is a literal which is allowed to be resolved upon, and $A_1\Theta$ is an instance of more than one A_i, then $A_1\Theta$ occurs in a false clause, because true clauses are multisets. Because the A-order is liftable one of the A_i can be factored upon.

Chapter 3

The Superposition Calculus

The main goal of this chapter is to introduce the *superposition calculus*. Before we do this, we introduce two other approaches to saturation-based theorem proving with equality. The first one is *explicit axiomatization*, the second is *paramodulation*. We explain why these two approaches are unsatisfactory, and how superposition improves on them. We also prove the completeness of superposition.

3.1 Introduction

In this document we discuss approaches for saturation-based theorem proving for first-order logic with equality. In the first section, we discuss a couple of approaches. We start with a very simple one, and approach with the best known strategy today, *the superposition calculus*. It is the best approach, because it has the most restrictive forward reasoning rules, combined with the strongest deletion and simplification rules. The completeness is proven in a single proof. This proof provides an abstract deletion principle. The clause deletion rules that are in use today are instances of this abstract deletion principle. Because of this, there is no need to repeat the completeness proof for different deletion principles.

We will explain why each next approach improves over the previous one. The first approach is explicit axiomatization of equality. The last approach is the *superposition calculus* from Bachmair and Ganzinger. This is the best known approach for theorem proving with equality at this moment. The completeness proof of the superposition calculus is able to give a unified explanation of all known important optimizations for theorem proving with equality.

3.1.1 Axiomatization of Equality

As a first approach, observe that can simply axiomatize the properties of equality, and use a resolution strategy without equality on the result. Equality can be axiomatized by the following axioms:

EQREFL $\forall x\ (x \approx x)$.

EQTRANS $\forall xyz\ (x \approx y \wedge y \approx z \rightarrow x \approx z)$.

EQSYM $\forall xy\ (x \approx y \rightarrow y \approx x)$.

EQREPL For each predicate symbol p with arity n, the following axiom has to be added:

$$\forall x_1 y_1 \cdots x_n y_n\ (\ x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n \wedge p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n)\ ).$$

EQFUNC For each function symbol f with arity n, the following axiom has to be added:

$$\forall x_1 y_1 \cdots x_n y_n\ (\ x_1 \approx y_1 \wedge \cdots \wedge x_n \approx y_n \rightarrow f(x_1, \ldots, x_n) \approx f(y_1, \ldots, y_n)\ ).$$

It is easily seen that resolvents between the equality axioms, and resolvents betwee the axioms and the problem clauses can produce enormous amounts of clauses. Dependent which side of the EQREPL-axioms is maximal, either all possible ground equalities are produced, or the EQREPL axioms can resolve with every equality in the problem. In both cases, the impact on efficiency is catastrophic. For this reason, we introduce a dedicated rule for equality in the next section, so that that the EQREPL-axioms can be removed.

3.1.2 Paramodulation

In natural deduction, there is the principle of replacement of equals by equals: If one has derived the equality $t_1 \approx t_2$, one may freely replace occurrences of t_1 by t_2 in other formulas. If one carries over this principle to saturation-style theorem proving, one obtains the *the paramodulation rule*. [1] The main difference with natural deduction is the fact that it is not sufficient to look for exact copies of t_1. Instead one has to look for terms that are unifiable with t_1.

Definition 3.1.1 *The* paramodulation rule *is the following rule:*
If $c_1 = [t_1 \approx t_2] \cup R_1$ *and* $c_2 = [A[t_1']\ ] \cup R_2$ *are clauses, such that* t_1' *and* t_1 *are unifiable, and* Θ *is the mgu, then the clause* $[A\Theta[t_2\Theta]\ ] \cup R_1\Theta \cup R_2\Theta$ *is a* paramodulant *of* c_1 *and* c_2. *The clause* c_1 *is called the* from-*clause, and clause* c_2 *is called the* into-*clause.*

We give some examples:

Example 3.1.2

[1] It seems to be tradition in ATP to give meaningless names to rules. Other examples are factoring, and demodulation

- *The clauses* $c_1 = [1 + 1 \approx 2]$ *and* $c_2 = [\sin(X)^2 + \cos(X)^2 \approx 1]$ *have the following paramodulants.*
  c_1 *into* c_2 :
  $$[\sin(X)^{1+1} + \cos(X)^2 \approx 1],$$
  $$[\sin(X)^2 + \cos(X)^{1+1} \approx 1],$$
  c_1 *into itself:*
  $$[2 \approx 2],$$
  $$[1 + 1 \approx 1 + 1],$$
  c_2 *into* c_1 :
  $$[\sin(X)^2 + \cos(X)^2 + 1 \approx 2],$$
  $$[1 + \sin(X)^2 + \cos(X)^2 \approx 2],$$
  c_2 *into itself:*
  $$[\sin(X)^2 + \cos(X)^2 \approx \sin(Y)^2 + \cos(Y)^2],$$
  $$[1 \approx 1].$$
- *Consider clauses* $c_1 = [\neg nat(X), X + 0 \approx X]$ *and* $c_2 = [\neg nat(Y), 0 + Y \approx Y]$. *When using* c_1 *as from clause, the variable* X *is unifiable with every subterm of* c_2 :
  $$[\neg nat(X), \neg nat(X + 0), 0 + X \approx X],$$
  $$[\neg nat(0), \neg nat(y), (0 + 0) + Y \approx Y],$$
  $$[\neg nat(X), 0 + (X + 0) \approx X].$$

The unrestricted paramodulation rule is a large improvement over explicit axiomatization, but it is still quite inefficient, as can be from the last example.

The superposition calculus further improves on paramodulation in the following ways.

- An order will be used which controls the direction in which equalities in from-clauses are used. Using the order, equalities need to be used only from the bigger to the smaller side. In the previous example, using superposition, the equality $0 + Y \approx Y$ needs to be applied only in the direction $0 + Y \Rightarrow Y$.
- Within a clause, only the maximal atom needs to be paramodulated into or from. For example, in the first clause $[\neg nat(X), \neg nat(X + 0), 0 + X \approx X]$, the atom $\neg nat(X + 0)$ could be maximal. In that case, the equality $0 + X \approx X$ will not be used at all.

- Many useful clause deletion and simplification strategies become available. For example, the clause $[s(a) \approx s(b)]$ can be deleted in the presence of $a \approx b$.

  In the presence of $[0 + X \approx X]$, the clause $[0 + (0 + X) \approx X]$ can be simplified into $[0 + X \approx X]$. This clause can be simplified one more time, which results in $[0 \approx 0]$. This clause is a tautology, and it can be completely removed.

In addition, the superposition solves a problem with paramodulation, namely that it explains why the EQFUNC-axioms can be deleted, and why paramodulation into variables can be avoided.

It is easy to see that the paramodulation rule makes the EQTRANS, the EQSYMM and the EQREPL-axioms unnecessary. It is however not so easy to see that the EQFUNC axioms can be deleted. The problems are due to the fact that paramodulation steps cannot always be *lifted*. For resolution without equality, completeness of a restriction is usually proven by first proving completeness for ground clauses, and after that proving that every ground proof can be simulated by a non-ground proof.

Unfortunately, lifting does not work with paramodulation because it is sometimes impossible to eproduce paramodulation steps that take place in instantiated terms.

Example 3.1.3 *(Failure of lifting in the context of paramodulation) Consider the ground paramodulation step:*

$$[p(s(a)), q(s(a))],\ [a \approx b] \Rightarrow [p(s(b)), q(s(a))].$$

If $[p(s(a)), q(s(a))]$ *is replaced by the non-ground clause* $[p(X), q(X)]$*, it is not possible to derive a clause that has* $[p(s(b)), q(s(a))]$ *as instance, when only paramodulation is used.*

Lifting can be restored if one keeps the EQFUNC-axioms, but this is again very inefficient, because the EQFUNC-axiom can cause enumeration of all equalities. In the example, one could resolve $a \approx b$ with $\forall xy\ x \approx y \rightarrow s(x) \approx s(y)$. The result is $s(a) \approx s(b)$. This clause can paramodulate into $\{p(X), q(X)\}$, and we derive $\{p(s(b)), q(s(a))\}$.

3.2 Ground Rewrite Systems

In the rest of this document we will prepare for introducting the superposition calculus and proving its completeness. The completeness proof is in its structure analogeous to the completenss proof for resolution without equality.
If a resolution prover tries to derive the empty clause, and does not succeed, it works towards an infinite set in which all inferences that can be made, have been made. Such a set is called *saturated set*. In the completeness proof, we will show that every saturated set which does not contain the empty clause, has a model.

The model will be represented by a rewrite system. In the non-equality case, a interpretation can be represented by a set of ground atoms. An atom is true in the interpretation iff it occurs in the set. This will not be enough for first-order logic with equality, because we have to take into account that equal terms have to be interpreted by the same object, and that atoms involving equal terms should receive the same truth value.

For this reason, we will represent interpretations by ground rewrite systems. A *rewrite system* evaluates terms by making equality replacements in a controlled way. The rewrite system can be designed in such a way that terms that are equal will be replaced into the same term in a finite number of steps. The rewrite system can also be used for assigning the truth values to atoms. Atoms that are true in the interpretation will eventually be replaced a designated constant **true**.

Definition 3.2.1 *A* rewrite system $\mathcal{R}$ *is a set of form*

$$\{t_1 \rightarrow u_1, \quad t_2 \rightarrow u_2, \quad t_3 \rightarrow u_3, \quad \dots\}.$$

The rewrite system need not be finite. Each t_i, u_i has to be a ground term. Each object $t \rightarrow u$ is called a rewrite rule *(or just* rule*)*

Let $\mathcal{R}$ be a rewrite system. We define the one-step rewrite relation $\Rightarrow$ *as follows: If t is a term, and $\mathcal{R}$ contains a rule $t' \rightarrow u'$, such that t' is a subterm of t, and the term u is obtained by replacing one occurrence of t' by u' in t, then $t \Rightarrow u$. The iterated rewrite relation $\Rightarrow^*$ is recursively defined as follows:*

- *For each term t, $\quad t \Rightarrow^* t$.*
- *If $t \Rightarrow^* u$ and $u \Rightarrow^* v$, then $t \Rightarrow^* v$.*

Let t be a term. We say that t is in normal form *if there is no u, such that $t \Rightarrow u$. We say that u is a* normal form *of t if $t \Rightarrow^* u$ and u is in normal form.*

Example 3.2.2 *Let $\mathcal{R} = \{a \rightarrow b, \quad b \rightarrow c\}$.*

On the term $f(a, a)$, two rewrites are possible, so we have

$$f(a, a) \Rightarrow f(a, b), \text{ and } f(a, a) \Rightarrow f(b, a).$$

On the term $f(a, b)$, also two rewrites are possible:

$$f(a, b) \Rightarrow f(b, b), \text{ and } f(a, b) \Rightarrow f(a, c).$$

The term $f(c, c)$ is in normal form, because no rewrites are possible. The term $f(c, c)$ is a normal form of $f(a, a)$.

It can be seen that in this rewrite system, every term has exactly one normal from. This is a very desirable property, because otherwise, possibly the result of a computation would not be well-defined.

Our final aim is to use rewrite systems for representing interpretations. Given a rewrite system, the domain of the interpretation consists of the set of possible normal forms of terms. The ground atoms that are true in the interpretation will be the atoms $p(t_1, \dots, t_n)$ for which $p(t_1, \dots, t_n) \Rightarrow^*$ **true**.

Example 3.2.3 *Consider the ground clause set*

$$C = \quad \{p(a)\}, \quad \{a \approx b\}, \quad \{a \approx c\}, \quad \{\neg q(c)\}.$$

C is true in the interpretation defined by the rewrite system

$$\mathcal{R} = \{ \ a \rightarrow c, \quad b \rightarrow c, \quad p(c) \rightarrow \mathbf{true} \ \}.$$

*For the atom $p(a)$, we have $p(a) \Rightarrow^*_{\mathcal{R}}$ **true**. For the equality, $a \approx b$, we have $a \Rightarrow^*_{\mathcal{R}} c$, and $b \Rightarrow^*_{\mathcal{R}} c$, so that it is true. For $q(c)$, we have $q(c) \not\Rightarrow^*_{\mathcal{R}}$ **true**, so that $\neg q(c)$ is true.*

It can be shown that for every clause set, each of its interpretations can be represented by a rewrite system. On the other hand, not every rewrite system represents an interpretation. This is caused by possible problems with the normal forms. There are two problematic situations possible:

1. A term t has no normal form at all.
2. A term t has more than one normal form.

As an example of (1), consider the rewrite system $\{0 \rightarrow s(0)\}$. In this system, the term 0 has no normal form, because there exists the infinite sequence

$$0 \Rightarrow s(0) \Rightarrow s^2(0) \Rightarrow s^3(0) \Rightarrow s^4(0) \Rightarrow \cdots$$

As an example of (2), consider the rewrite system $\{a \rightarrow b, \ a \rightarrow c\}$. In this rewrite system, the term a has two normal forms b and c.

Since we want to use rewrite systems for defining interpretations, it is clear that we need to impose some additional restictions, in order to ensure that every term and every atom has exactly one normal form.

We will discuss the problem of non-termination in the following section. We will introduce a complexity measure on terms, and assume that the rules of the rewrite systems always replace bigger by smaller terms.

In order to deal with the problem of more-than-one normal form, we will introduce the notion of *confluence*. Confluence means, that whenever a term can be rewritten into two different terms, it is possible to rewrite them into a single term again. As an example, consider the rewrite system of Example 3.2.2. The term $p(a, a)$ can be rewritten into two distinct terms $p(b, a)$ and $p(a, b)$, but both can be rewritten again into the term $p(b, b)$. Confluence will be discussed in Section 3.2.3.

3.2.1 Reduction Orders

We want to use reduction orders in order to ensure that every term has at least one normal form. Formall, this can be interpreted in two different ways. One possibility is to require that from every term at least one normal form is reachable. The second possibility is to require that from every term, every rewrite sequence will reach a normal form in a finite number of steps. It is clear that the second property is more desirable, because it implies that one make rewrites without thinking, and still is guaranteed to reach a normal form.

Definition 3.2.4 *A rewrite system $\mathcal{R}$ is* weakly normalizing *if for every term t, there exists a term u, such that $t \Rightarrow^*_{\mathcal{R}} u$, and u is in normal form.*

A rewrite system $\mathcal{R}$ is strongly normalizing *if every rewrite sequence $t_1 \Rightarrow_{\mathcal{R}} t_2 \Rightarrow_{\mathcal{R}} t_3 \Rightarrow_{\mathcal{R}} \cdots$ is finite.*

Strongly normalizing is the property that we are interested in. One possiblity to show that a rewrite system is strongly normalizing, is by showing that in every rewrite sequence $t_0 \Rightarrow t_1 \Rightarrow t_2 \Rightarrow \cdots$, the sizes of the terms are strictly decreasing. This often can be done by showing that in $\mathcal{R}$, for each rule $(u \rightarrow v) \in \mathcal{R}$, the left hand side u is strictly bigger than v.

If we want to ensure that two terms t_1 and t_2 are equal in the interpretion, we simply add the rule $t_1 \rightarrow t_2$, or $t_2 \rightarrow t_1$ to the rewrite system, dependent on which term is bigger. For example, if we want $s(0) \approx 0$ to be true, one should add the rewrite rule $s(0) \rightarrow 0$, and not $0 \rightarrow s(0)$.

Unfortunately, complexity measures cannot decide an ordering between all terms. For example, the terms a, b or $s(0), s(1)$ have the same complexity. In order to make sure that all terms can be compared, we move to the abstract notion of reduction order:

Definition 3.2.5 *A* reduction order *is a relation $\succ$ on terms satisfying the following conditions:*

O1 $\forall x\ (x \not\succ x)$.

O2 $\forall xyz\ (x \succ y \wedge y \succ z$ *implies* $x \succ z)$.

O3 $\forall xy\ (x \succ y)$ *or* $(y \succ x)$ *or* $(x = y)$.

WF *In every set of terms T over a finite signature there is at least one minimal term t. This is a term $t \in T$, such that there is no $t' \in T$ with $t \succ t'$.*

CONT *Relation $\succ$ is preserved in contexts: If $x \succ y$, then $t[x] \succ t[y]$.*

T *The truth constant* **true** *is minimal: If $x \neq$* **true**, *then $x \succ$* **true**.

The properties O1, O2, WF, CONT would be satisfied by the order based on term complexity. T could be satisfied by giving **true** a term complexity of 0. It is O3 where term complexity fails.

The most important reduction orders for theorem proving are the family of *Knuth-Bendix* orders. They are obtained by taking the complexity-order as starting point, and completing the order by inserting some arbitrary order on terms with the same complexity.

Other common reduction orders are *recursive path* orders, or *lexicographic path* orders. These orders are more complicated, but they are superior on equality theories containing distributivity axioms. We now show that reduction orders can be used for ensuring termination.

Theorem 3.2.6 *Let $\mathcal{R}$ be a rewrite system. Let $\succ$ be a reduction order. Assume that for every rule $u \rightarrow v$ in $\mathcal{R}$, we have $u \succ v$. Then $\mathcal{R}$ is strongly normalizing.*

Proof
Let $t_0 \Rightarrow_{\mathcal{R}} t_1 \Rightarrow_{\mathcal{R}} t_2 \Rightarrow_{\mathcal{R}} t_3 \Rightarrow_{\mathcal{R}} \cdots$ be a rewrite sequence based on $\mathcal{R}$. For each i, there is a rule $(u \rightarrow v) \in \mathcal{R}$, such that t_i can be written as $t_i[u]$ and t_{i+1} can be written as $t_i[v]$. Since $u \succ v$ and $\succ$ has property CONT, it follows that $t_i \succ t_{i+1}$. Hence t_0, t_1, t_2 is a decreasing sequence. Because $\succ$ is WF, it must be a finite sequence.
Because of O3, one can always direct all rules in $\mathcal{R}$ in such a way that they are decreasing. In the next section, we introduce Knuth-Bendix orders. In this section, we conclude by showing that the reduction relation of strongly normalizing rewrite systems can be used for induction proofs. We need this property in Section 3.2.3.

Theorem 3.2.7 *A rewrite system $\mathcal{R}$ is strongly normalizing iff in every non-empty set of terms T, there is a term t such that all t' for which $t \Rightarrow_{\mathcal{R}}^{+} t'$ are not in T.*

Proof
Assume that $\mathcal{R}$ is strongly normalizing.

Let T be a non-empty set. Let t_0 be an element in T. Construct a rewrite sequence starting with t_0. Because $\mathcal{R}$ is strongly normalizing, the sequence must be finite.

$$t_0 \Rightarrow_{\mathcal{R}} t_1 \Rightarrow_{\mathcal{R}} t_2 \Rightarrow_{\mathcal{R}} t_3 \Rightarrow_{\mathcal{R}} \cdots \Rightarrow_{\mathcal{R}} t_n.$$

Because the sequence is finite, there is a last t_i which is still in T. This is the desired t'.

As for the other direction, assume that in every non-empty set of terms T, there is a term t, such that all terms t' for which $t \Rightarrow_{\mathcal{R}}^{+} t'$, are not in T. We have to show that $\mathcal{R}$ is strongly normalizing.

Let t_0 be an arbitrary term. Let

$$t_0 \Rightarrow_{\mathcal{R}} t_1 \Rightarrow_{\mathcal{R}} t_2 \Rightarrow_{\mathcal{R}} t_3 \Rightarrow_{\mathcal{R}} t_4 \Rightarrow_{\mathcal{R}} \cdots$$

be a rewrite sequence starting with t_0. Let $T = \{t_0, t_1, t_2, t_3, \ldots\}$. This set is non-empty because it contains at least t_0. Hence there must be a term $t_i \in T$, s.t. for every term t' with $t_i \Rightarrow_{\mathcal{R}}^{+} t'$, $t' \notin T$. Assume such a t_i is given. If there would exist a t_{i+1} with $t_i \Rightarrow_{\mathcal{R}} t_{i+1}$, then it would follow that $t_{i+1} \notin T$. Hence t_i must be the last term of the rewrite sequence.

3.2.2 Knuth-Bendix Orders

Knuth-Bendix orders are obtained when an order based on complexity is made total using some other order. If one has a finite signature, then each set of ground terms with a given complexity is finite. Within such a set, one can use any order to make the order total. We will assume that the signature, (which is finite), has a fixed order $\sqsupset$, and use the standard dictionary ordering on the prefix representations of the terms.

Definition 3.2.8 *Let $\#$ be a function that assigns non-negative integers to the symbols in the signature, with the exception of* **true**, *to which it assigns* 0. *The function $\#$ can be extended to terms as follows:*

- *For a term a $f(t_1, \ldots, t_n)$, $\quad \#f(t_1, \ldots, t_n) = \#f + \#t_1 + \cdots + \#t_n$.*

Definition 3.2.9 *Let $f(t_1, \ldots, t_n)$ be a term. The* prefix representation *$\pi(f(t_1, \ldots, t_n))$ of $f(t_1, \ldots, t_n)$ is defined as the sequence*

$$(f) \cdot \pi(t_1) \cdot \ldots \cdot \pi(t_n).$$

The prefix representation of $p(a, b)$ is the sequence (p, a, b). The prefix representation of $p(s(0), s(0))$ is $(p, s, 0, s, 0)$.

Definition 3.2.10 *Let $\sqsupset$ be an order on the signature. We define the lexicographic extension of $\sqsupset$ to terms, $\sqsupset_{term}$, as follows:*

$$t_1 \sqsupset_{term} t_2 \text{ iff } \pi(t_1) \sqsupset_{lex} \pi(t_2).$$

Here $\sqsupset_{lex}$ is the standard lexicographic extension of $\sqsupset$ to sequences.

The lexicographic order $\sqsupset_{lex}$ is the order with which telephone books and dictionaries are ordered. The following property is standard.

Lemma 3.2.11 *$\sqsupset_{lex}$ is a total order, i.e. it satisfies O1, O2 and O3.*

As a consequence, also

Lemma 3.2.12 *$\sqsupset_{term}$ satifies O1, O2, O3. In addition, it satisfies CONT.*

Definition 3.2.13 *The* Knuth-Bendix *order $\succ_{KBO}$ is defined as follows: $t_1 \succ_{KBO} t_2$ iff either $\#t_1 > \#t_2$, or $\#t_1 = \#t_2$ and $t_1 \sqsupset_{term} t_2$.*

Theorem 3.2.14 *$\succ_{KBO}$ is a reduction order.*

Proof
We prove O1,O2,O3,WF,CONT,T:

O1 $t \succ_{KBO} t$ would imply that either $\#t > \#t$, or $\#t = \#t$ and $\#t \sqsupset_{term} \#t$. The first case is clearly impossible, the second case is also impossible because $\sqsupset_{term}$ has property O1.

O2 Assume that $t_1 \succ_{KBO} t_2$ and $t_2 \succ_{KBO} t_3$. If either $\#t_1 > \#t_2$, or $\#t_2 > \#t_3$, then $\#t_1 > \#t_3$, and $t_1 \succ_{KBO} t_3$.

Otherwise, it must be the case that $\#t_1 = \#t_2 = \#t_3$. Then, it follows from the fact that $\sqsupset_{term}$ has property O2, that $t_1 \sqsupset_{KBO} t_3$.

O3 If $\#t_1 \neq \#t_2$, then it is clear that O3 holds. Otherwise, $\#t_1 = \#t_2$. In that case, O3 follows from the fact $\sqsupset_{term}$ has property O3.

WF Let T be a finite, non-empty set of terms over a finite signature. Let s be the smallest complexity $\#t$ of a term $t \in T$. Let T' be the subset of T with complexity s. Because the signature is finite, T' must be finite in size. The order $\sqsupset_{term}$ is a total order on T'. Hence, there is a minimal element in T'. This is also the minimal element of T.

T Because **true** is the only term with complexity 0.

Example 3.2.15 *Put $\#a = \#b = \#c = \#f = 1$. Put $a \sqsupset b \sqsupset c \sqsupset f$ (as symbols). Then $a \succ_{KBO} b$, and $b \succ_{KBO} c$. Also $f(a,a) \succ_{KBO} a$, because of complexity. The terms $f(a,a)$ and $f(a,b)$ have the same complexity, but $f(a,a) \succ_{KBO} f(a,b)$ because of the lexicographic extension of $\sqsupset$.*

3.2.3 Confluence

In the previous sections, we have found ways of ensuring that every term has at least one normal form. Now we discuss the problem of ensuring that there is at most one normal form.

Definition 3.2.16 *A rewrite system $\mathcal{R}$ is called* confluent *(or* Church-Rosser*) if*

Whenever $t \Rightarrow^+_{\mathcal{R}} u_1,\ t \Rightarrow^+_{\mathcal{R}} u_2,$ there exists a v with $u_1 \Rightarrow^*_{\mathcal{R}} v$ and $u_2 \Rightarrow^*_{\mathcal{R}} v$.

A rewrite system $\mathcal{R}$ is called weakly confluent *or* weakly Church-Rosser *if*

Whenever $t \Rightarrow_{\mathcal{R}} u_1,\ t \Rightarrow_{\mathcal{R}} u_2,$ there exists a v with $u_1 \Rightarrow^*_{\mathcal{R}} v$ and $u_2 \Rightarrow^*_{\mathcal{R}} v$.

Lemma 3.2.17 *Confluence can be alternatively formulated as*

Whenever $t \Rightarrow^*_{\mathcal{R}} u_1,\ t \Rightarrow^*_{\mathcal{R}} u_2,$ there exists a v with $u_1 \Rightarrow^*_{\mathcal{R}} v$ and $u_2 \Rightarrow^*_{\mathcal{R}} v$.

Lemma 3.2.18 *Let $\mathcal{R}$ be a rewrite system that is confluent. Then every term t has at most one normal form.*

Proof
Assume that $t \Rightarrow^*_{\mathcal{R}} u_1,\quad t \Rightarrow^*_{\mathcal{R}} u_2$ and both u_1, u_2 are normal forms. Then, because $\mathcal{R}$ is confluent, there exists a v, s.t. $u_1 \Rightarrow^*_{\mathcal{R}} v$ and $u_2 \Rightarrow^*_{\mathcal{R}} v$. Since both u_1 and u_2 are normal forms, it must be the case that $u_1 = v$ and $u_2 = v$. Hence $u_1 = u_2$.

It is easily seen that every confluent rewrite system is also weakly confluent. Conversely, a weakly confluent rewrite system need not be confluent in case it has infinite reductions. However, when the rewrite system has no infinite reductions, then weak confluence implies confluence.

Theorem 3.2.19 *Let $\mathcal{R}$ be a strongly normalizing rewrite system. If $\mathcal{R}$ is weakly confluent, then $\mathcal{R}$ is strongly confluent.*

Proof

We will make use of Theorem 3.2.7. Let $\mathcal{BAD}$ be defined as

$$\mathcal{BAD} = \{s \mid \exists t_1 t_2 \ s \Rightarrow^*_{\mathcal{R}} t_1 \land s \Rightarrow^*_{\mathcal{R}} t_2 \land \neg\exists u \ (\ t_1 \Rightarrow^*_{\mathcal{R}} u \land t_2 \Rightarrow^*_{\mathcal{R}} u \ ) \ \}.$$

(These are the s for which strong confluence fails)

We will assume that $\mathcal{BAD}$ is not empty and derive a contradiction. Suppose that $\mathcal{BAD}$ is not empty. By Theorem 3.2.7, there is an element $b \in \mathcal{BAD}$ such that $\forall b'$ ($b \Rightarrow^+_{\mathcal{R}} b'$ implies $b' \notin \mathcal{BAD}$). By definition of $\mathcal{BAD}$, there exist t_1, t_2, such that

$$b \Rightarrow^*_{\mathcal{R}} t_1 \text{ and } b \Rightarrow^*_{\mathcal{R}} t_2 \text{ and } \neg\exists u \ (t_1 \Rightarrow^*_{\mathcal{R}} u \text{ and } t_2 \Rightarrow^*_{\mathcal{R}} u \ ).$$

If $t_1 = b$, then it is possible to take $u = t_2$. In addition, if $t_2 = b$, it is possible to take $u = t_1$. Therefore, $b \neq t_1$ and $b \neq t_2$. Now let b_1, b_2 be possible first elements on the reduction sequences $b \Rightarrow^+_{\mathcal{R}} t_1$ and $b \Rightarrow^+_{\mathcal{R}} t_2$, so that we have

$$b \Rightarrow_{\mathcal{R}} b_1 \Rightarrow^*_{\mathcal{B}} t_1 \text{ and } b \Rightarrow_{\mathcal{R}} b_2 \Rightarrow^*_{\mathcal{B}} t_2.$$

Since $\mathcal{R}$ is weakly confluent, there exists a term v, s.t.

$$b_1 \Rightarrow^*_{\mathcal{R}} v \text{ and } b_2 \Rightarrow^*_{\mathcal{R}} v.$$

Because $b_1 \notin \mathcal{BAD}$, $b_1 \Rightarrow^*_{\mathcal{R}} t_1$, and $b_1 \Rightarrow^*_{\mathcal{R}} v$, there exists a term w_1, s.t.

$$t_1 \Rightarrow^*_{\mathcal{R}} w_1 \text{ and } v \Rightarrow^*_{\mathcal{R}} w_1.$$

Similarly, there exists a term w_2, s.t.

$$t_1 \Rightarrow^*_{\mathcal{R}} w_2 \text{ and } v \Rightarrow^*_{\mathcal{R}} w_2.$$

Since also $v \notin \mathcal{BAD}$, $v \Rightarrow^*_{\mathcal{R}} w_1$ and $v \Rightarrow^*_{\mathcal{R}} w_2$, there must exist a further x, s.t.

$$w_1 \Rightarrow^*_{\mathcal{R}} x \text{ and } w_2 \Rightarrow^*_{\mathcal{R}} x.$$

We show that this x contradicts the original assumption that $b \in \mathcal{BAD}$. Since we have

$$t_1 \Rightarrow^*_{\mathcal{R}} w_1 \Rightarrow^*_{\mathcal{R}} x, \text{ and } t_2 \Rightarrow^*_{\mathcal{R}} w_2 \Rightarrow^*_{\mathcal{R}} x,$$

we have $t_1 \Rightarrow^*_{\mathcal{R}} x$ and $t_2 \Rightarrow^*_{\mathcal{R}} x$.

Since we are only using strongly normalizing rewrite systems, we can freely use Theorem 3.2.19.

Definition 3.2.20 *Let $\mathcal{R}$ be a rewrite system. A* critical pair *is a pair of distinct rules $(u_1 \to v_1) \in \mathcal{R}$, $(u_2 \to v_2) \in \mathcal{R}$, s.t. either $u_1 = u_2$ or u_1 is a subterm of u_2.*

Definition 3.2.21 *Let $\mathcal{R}$ be a rewrite system. Let $(u_1 \to v_1)$, $(u_2 \to v_2)$ be a critical pair of $\mathcal{R}$.*

Since u_1 is a subterm of u_2, we can write u_2 in the form $u_2[u_1]$. Then $u_2[u_1]$ can be rewritten in two possible ways, the possible results are $u_2[v_1]$ and v_2. The critical pair is mergable *if there exists a term w, s.t.*

$$u_2[v_1] \Rightarrow^*_{\mathcal{R}} w \text{ and } v_2 \Rightarrow^*_{\mathcal{R}} w.$$

Theorem 3.2.22 *Let $\mathcal{R}$ be a rewrite system. If every critical pair of $\mathcal{R}$ is mergable, then $\mathcal{R}$ is weakly confluent.*

Proof
Let t, u_1, u_2 be terms such that $t \Rightarrow_{\mathcal{R}} u_1$ and $t \Rightarrow_{\mathcal{R}} u_2$.

Let $v_1 \rightarrow w_1$ and $v_2 \rightarrow w_2$ be the rules with which the rewrite steps are made.

There are two possibilities.

1. The two rewrite steps are made at incomparable positions. In that case, one can write

$$t = t[v_1, v_2], \quad u_1 = t[w_1, v_2], \quad u_2 = t[v_1, w_2].$$

   Both terms u_1, u_2 can be rewritten into $t[w_1, w_2]$.

2. One of the rewrite steps is made at a subposition of the other rewrite step. Assume w.l.o.g. that the first rewrite step is made at a subposition of the second rewrite step. One can write

$$t = t[v_2[v_1]], \quad u_1 = t[v_2[w_1]], \quad u_2 = t[w_2].$$

   The rules $v_1 \rightarrow w_1$ and $v_2 \rightarrow w_2$ form a critical pair. Hence it is mergable. This means that the terms $v_2[w_1]$ and w_2 must be mergable into a term z. As a consequence both $t[v_2[w_1]]$ and $t[w_2]$ can be rewritten into $t[z]$.

Example 3.2.23 *The rewrite system $\mathcal{R}$ in definition 3.2.2 has no critical pairs. Therefore, it is weakly confluent.*
If one takes the Knuth-Bendix order with $a \succ b \succ c$, the rules of $\mathcal{R}$ are strictly decreasing. Therefore, $\mathcal{R}$ is strongly terminating. By Theorem 3.2.19, it is strongly confluent.

This means that $\mathcal{R}$ is able to define an interpretation.

If a rewrite system $\mathcal{R}$ has a critical pair consisting of rules $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$, that is not mergable, then one can make it mergable by adding either $v_1 \rightarrow v_2$ or $v_2 \rightarrow v_1$. Doing this eagerly, is called *completion.* In the ground case, completion is guaranteed to terminate, and result in a weakly confluent rewrite system. The completeness proof of the superposition calculus constructs rewrite systems without critical pairs.

3.3 The Ground Superposition Calculus

We define the ground superposition calculus. In the completeness proof, an interpretation will be constructed, which is represented by a rewrite system that is directed by a reduction order.

In the calculus, the reduction order can be used for directing equalities, so that equalities are used only from greater to smaller terms. In addition, only the greatest literal in a clause needs to be rewritten into.

Definition 3.3.1 *Let $\succ$ be a reduction order, defined on terms. The* extension *of $\succ$ to literals $\succ_{lit}$ is defined through the following mapping μ from literals to multisets.*
For a positive equality $(t_1 \approx t_2)$, let $\mu(t_1 \approx t_2) = [[t_1], [t_2]]$. For a negative equality $(t_1 \not\approx t_2)$, let $\mu(t_1 \not\approx t_2) = [[t_1, t_2]]$.

Then $A_1 \succ_{lit} A_2$ if $\mu(A_1) \succ^2 \mu(A_2)$, where $\succ^2$ is the multiset extension of the multiset extension of $\succ$.

Note that μ is defined only for equality atoms. Since we assume that a atom $p(t_1, \ldots, t_n)$ if $p(t_1, \ldots, t_n) \Rightarrow^* \textbf{true}$, we can as well replace $p(t_1, \ldots, t_n)$ by the equality $p(t_1, \ldots, t_n) \approx \textbf{true}$. Similarly, the negative literal $\neg p(t_1, \ldots, t_n)$ can be replaced by $p(t_1, \ldots, t_n) \not\approx \textbf{true}$. This has the advantage that the calculus can be somewhat simplified.

The extension of $\succ$ to literals needs to satisfy two conditions: The first is that every equality is smaller than every literal that it potentially rewrites. The second is that the equality factoring rule must be decreasing. Using the double multiset order is only one of the possible ways to obtain this.

Within clauses, $\prec_{lit}$ is used for determining which equality will be paramodulated from, and which equality will be paramodulated into. In addition to $\prec_{lit}$, it is also possible to 'switch off' $\prec_{lit}$, and instead use a negative literal. For this purpose, we introduce selection functions. A selection function can either follow the order $\succ_{cls}$, or override the order and select a negative literal.

Definition 3.3.2 *Remember that clauses are represented by multisets of literals. A function Σ from non-empty multisets of literals to literals is a* a selection function based on *a given reduction order $\succ$, if always $\Sigma(c) \in c$, and in addition*

- *$\Sigma(c)$ is the $\succ_{lit}$-maximal atom of c, or*
- *$\Sigma(c)$ is a negative atom.*

If A is a literal, and $\Sigma(c) = A$, we way that A is the selected literal of *c.*

The superposition calculus consists of forward reasoning rules, and of a general redundancy criterion. The forward reasoning rules determine which clauses have to be added, in order to be complete. The redundancy criterion determines on which condition a clause can be deleted without loosing completeness. We will see that the redundancy criterion covers all of the standard clause elimination principles. We first introduce the forward reasoning rules:

equality reflexivity Let $c = [t \not\approx t] \cup R$ be a clause, s.t. $t \not\approx t$ is selected. Then R is obtained by *equality reflexivity* from c.

positive superposition Let $c = [t_1 \approx t_2] \cup R$ and $d = [u_1 \approx u_2] \cup S$ be clauses, and let u_1' be a term, s.t.

1. $(t_1 \approx t_2)$ is selected in c, and $t_1 \succ t_2$,
2. $(u_1 \approx u_2)$ is selected in d, and $u_1 \succ u_2$,

3. $u_1 \Rightarrow u_1'$ under the rewrite system $\{t_1 \rightarrow t_2\}$.

Then the clause $[u_1' \approx u_2] \cup R \cup S$ is obtained by *positive superposition from c into d.*

negative superposition Let $c = [t_1 \approx t_2] \cup R$ and $d = [u_1 \not\approx u_2] \cup S$ be a clauses, and let u_1' be a term, s.t.

1. $(t_1 \approx t_2)$ is selected in c, and $t_1 \succ t_2$,
2. $(u_1 \not\approx u_2)$ is selected in d, and $u_1 \succ u_2$,
3. $u_1 \Rightarrow u_1'$ under the rewrite system $\{t_1 \rightarrow t_2\}$.

Then the clause $[u_1' \not\approx u_2] \cup R \cup S$ is obtained by *negative superposition from c into d.*

equality factoring Let $c = [t_1 \approx t_2,\ \ t_1 \approx t_3] \cup R$ be a clause, s.t. $t_1 \succ t_2 \succeq t_3$, and $t_1 \approx t_2$ is selected in c. Then the clause $[t_1 \approx t_3,\ \ t_2 \not\approx t_3] \cup R$ is *an equality factor* of c.

In order to be able to define the abstract redundancy principle, the order $\succ_{lit}$ has to be extended to clauses.

Definition 3.3.3 *Let $\succ$ be a reduction order on terms. We define the extension $\succ_{cls}$ of $\succ$ to clauses as follows: Let $c = [A_1, \dots, A_p]$ and $d = [B_1, \dots, B_q]$ be clauses. We say that $c \succ_{cls} d$ iff*

$$[\ \mu(A_1), \dots, \mu(A_p)\ ] \succ^3 [\ \mu(B_1), \dots, \mu(B_q)\ ],$$

where $\succ^3$ is the triple multiset extension of $\succ$.

Definition 3.3.4 *Let C be a clause set. Let c be a clause. We call c* redundant *in C, if there exists a finite subset $D \subseteq C$, s.t. $D \models c$ and for each $d \in D$, $\ c \succ d$.*

In words, one can say that c is redundant in C if c is logically implied by smaller clauses in C.
The abstract redundancy cannot be used in practice, because for predicate logic, it is undecidable. However, it is possible to define subcases that are practical. Redundancy can be used in two ways: Firstly, one can use it for checking whether a given clause is redundant. If it is redundant, then it can be deleted. Secondly, one can try to simplify a given clause. If one can derive a new, smaller clause which logically implies the old clause, one can delete the old clause. We first give a few possible deletion rules, and after that a few simplification rules.

subsumption If $c \subset d$, and $c \in C$, then $d \succ c$ and $c \models d$. Hence d is redundant.

tautology elimination If the negative equalities in c imply the positive equalities in c, then $\models c$. Hence c is redundant.

demodulation If c has form $[t_1 \approx t_2] \cup R$, $t_1 \succ t_2$, and d has form $[\ A[t_1]\ ] \cup S$ with $R \subseteq S$, then d can be simplified into $[\ A[t_2]\ ] \cup S$.

equality reflexivity In the propositional case, equality reflexivity is a simplifiation rule.

In practice, it often happens that simplifying a clause results in a tautology, so that the clause can be completely deleted.
Most theorem provers do not implement the simplification rules in their full strength. Most provers demand in the case of demodulation that the equality clause is a unit clause. There is no simplification rule based on equality factoring, since $\{\neg\, t_2 \approx t_3, t_1 \approx t_3\} \cup R$ does not imply $\{t_1 \approx t_2, t_1 \approx t_3\} \cup R$.
There are many moments on which the simplification rules can be applied. When a clause, which has just been derived, is simplified, this is called *forward* simplification, (or forward unit-resolution, forward demodulation, etc.) When a kept clause is used to simplify other clauses, this is called *backward* simplification. In addition, one can make the following distinctions:

1. Always keep the clauses fully simplified. When a clause is created it is forward simplified. When a clause is kept, it is used to simplify the other clauses.

2. Keep only the clauses in **usable** fully simplified. When a clause is generated it is forward simplified. Kept clauses are not used for backward simplification.

3. Don't do any backward simplification at all. This is an acceptable option in many cases, and it is certainly easy to implement. However there is a risk of missing important simplifications.

The EQFACT-rule is provable using the other rules, but this does not imply that it is redundant.

Exercise 3.3.5 *Show that the EQFACT-rule is correct by showing that* $\Gamma = \{\{R, t_1 \approx t_2, t_1 \approx t_3\}, \{\neg\, R\}, \{t_2 \approx t_3\}, \{\neg\, t_1 \approx t_3\}\}$ *can be refuted by the other rules.*

3.3.1 Completeness

We will prove completeness of the ground superposition calculus. Informally, this means that when a certain clause set C is unsatisfiable, and the theorem prover generates sufficiently many clauses from C, that then it is guaranteed to eventually derive the empty clause. Adding sufficiently many clauses means: For every inference that is possible from C, for which the conclusion is not redundant, either the conclusion itself is added, or some simplification that makes the conclusion redundant.

The theorem prover systemstically needs to check all possible inferences, and check whether the conclusion is redundant. If the conclusion is not redundant, the theorem prover tries to simplify the conclusion and add it to the database. While the theorem prover does this, it approximates a set, in which all inference that can be made, have been made. Formally:

Definition 3.3.6 *Let C be a set of clauses. The set C is called* saturated *if every clause c that can be obtained from C by one of the rules equality reflexivity, positive/negative superposition, or equality factoring, is either present in C or redundant in C.*
Let J be a set of initial clauses. A clause set C is called a saturation *of J if C is a saturation and every clause $c \in J$ is either present or redundant in C.*

Theorem 3.3.7 *Let C be a saturated clause set. If C does not contain the empty clause, then C has a model.*

The rest of this section will be devoted to proving Theorem 3.3.7. The proof proceeds by transfinite induction, using $\succ_{cls}$ on C. So, in the rest of this section, assume that C is a given saturated clause set. The structure of the proof is as follows:

1. First, using transfinite recursion, we pass through the saturated clause set, and make some selected clause true, by picking an equality from them, and adding the equalities as rewrite rule to the interpretation.

2. Now the clauses from which we took an equality, are obviously true. For the others, we will show by transfinite induction, that they are also true. Here we use the fact that C is saturated. We will show for the other clauses, that it is possible to make an inference with them which results in a smaller clause. We will then by induction assume that this smaller clause is true, and prove from this that the original clause is true.

We define step (1), which makes some selected clauses true. In order to define a function f by transfinite recursion, one needs to give a procedure to obtain a value for $f(c)$, assuming that all c' with $c \succ c'$ already have a value $f(c')$.

Definition 3.3.8 *We define, by transfinite induction, for each clause $c \in C$, a rewrite system I_c as follows:*
Let c be a clause in C. Assume that for each d with $c \succ_{cls} d$, I_d is defined.
First define

$$J_c = \bigcup_{c \succ_{cls} d} I_d.$$

Next, if c is false in J_c, and the selected atom in c is a positive equality of form $t_1 \approx t_2$ with $t_1 \neq t_2$, then assume that $t_1 \succ t_2$. Now, if there is no equality $t_1 \approx t_3$ in c, such that J_c makes $t_2 \approx t_3$ true, and t_1 is in normal form with respect to J_c, then define $I_c := J_c \cup \{t_1 \to t_2\}$. Otherwise, define $I_c := J_c$.
Finally, put $I := \bigcup_{c \in C} I_c$.

The following theorem ensures that I is well-behaved, so that it can act as an interpretation for C.

Theorem 3.3.9 *The rewrite system I is strongly normalizing, and it is confluent.*

Proof
It is strongly normalizing, because for each rule $t_1 \rightarrow t_2$ in I, we have $t_1 \succ t_2$. It is confluent, because there exist no critical pairs in I. Suppose that $t_1 \rightarrow t_2$ and $u_1 \rightarrow u_2$ form a critical pair. Assume that $t_1 \rightarrow t_2$ was added while inspecting clause c. Assume that $u_1 \rightarrow u_2$ was added while inspecting clause d. Without loss of generality, one can assume that $d \succ_{cls} c$. Then J_d contains $t_1 \rightarrow t_2$. Because $u_1 \rightarrow u_2$ is added only in case that u_1 is in normal form with respect to J_d, it must be the case that t_1 did not occur in u_1. This contradicts the fact that $t_1 \rightarrow t_2$ and $u_1 \rightarrow u_2$ form a critical pair.

In Definition 3.3.8, those atoms that contributed to I are abviously true in I, because their atom $t_1 \approx t_2$ is true.

We will show that the remaining clause are true by transfinite induction on $\succ_{cls}$. The essence of the proof is based on showing that for the clauses that did not contribute to I, an inference is possible, resulting in a $\succ_{cls}$-smaller clause. Exactly what kind of inference is possible, depends on the reason why the clause was not allowed to contribute to I.
We first show another, very important property. At the moment that a clause $c = [t_1 \approx t_2] \cup R$ contributes to I, all atoms in R are false in J_c. We will show that later additions do not make R true, so that R is still false in I.

Theorem 3.3.10 *For every rewrite rule $t_1 \rightarrow t_2$ in I, there is a clause of form $[t_1 \approx t_2] \cup R$ in C, s.t. $t_1 \succ t_2$, the equality $t_1 \approx t_2$ is selected in $[t_1 \approx t_2] \cup R$, and R is false in I.*

Proof
Every rewrite rule $t_1 \rightarrow t_2$ in I is added due to some clause c of form $[t_1 \approx t_2] \cup R$ in C, for which $t_1 \succ t_2$ and $[t_1 \approx t_2]$ is selected in c.

Since c was false in J_c, also R must have been false in J_c. We will show for each literal in R that it is still false in I.

We first discuss the negative literals, then the positive literals. Let $u_1 \not\approx u_2$ be a negative equality in R. The fact that $u_1 \not\approx u_2$ is false in J_c means that u_1 and u_2 have a common normal form in J_c. Since $J_c \subseteq I$, u_1 and u_2 still have a common normal form in I.

Now let $u_1 \approx u_2$ be a positive equality in R. We first show that $u_1 \approx u_2$ is still false in I_c. We distinguish two cases, dependent on whether $u_1 \approx u_2$ contains t_1.

- If $u_1 \approx u_2$ contains t_1, then we may without loss of generality assume that u_1 contains t_1. It must be the case that $u_1 = t_1$, because otherwise $t_1 \approx t_2$ would not be maximal. If $t_1 \approx u_2$ were true in I_c, this means that t_1 and u_2 have a common normal form in I_c. The reduction sequence of t_1 must have form $t_1 \Rightarrow t_2 \Rightarrow \cdots$. Since $t_1 \succ t_2$ this implies that t_2 and u_2 already had a common normal form in J_c. This constradicts the construction in Definition 3.3.8.

- If u_1 and u_2 do not contain t_1, then both $t_1 \succ u_1, u_2$ by maximality of $t_1 \approx t_2$. Hence u_1 and u_2 are not affected by adding the rule $t_1 \rightarrow t_2$ and still have different normal forms in I_c.

It remains to show that $u_1 \approx u_2$ is still false in I. This can be done by showing that for every rewrite rule $v_1 \rightarrow v_2$, that is added due to a clause d for which $d \succ_{cls} c$, it must be the case that $v_1 \succ t_1$. Hence, it cannot rewrite u_1, u_2, nor a term derived from it. We use the totality of $\succ$.

First suppose that $v_1 = t_1$. This contradicts the fact that $v_1 \rightarrow v_2$ is only added on the condition that v_1 is in normal form.

Next suppose that $t_1 \succ v_2$. Because $v_1 \approx v_2$ is selected in d, it must be the maximal atom in d. As a consequence one would have $c \succ_{cls} d$.
As said, we will use transfinite induction to show that the clauses that did not directly contribute to I are also true. In most of the cases, we will show for a clause c, that a new d can be derived, for which $c \succ d$, on which induction can be applied, and that the truth of d implies the truth of c. In order to be able to use the induction, we need to show that some inferences are decreasing.

Theorem 3.3.11

1. *Assume that d is obtained from c by equality reflexivity. Then $c \succ_{cls} d$.*
2. *Assume that d is obtained by positive superposition from c_1 into c_2. Then $c_2 \succ_{cls} d$.*
3. *Assume that d is obtained by negative superposition from c_1 into c_2. Then $c_2 \succ_{cls} d$.*
4. *Assume that d is obtained by equality factoring from c. Then $c \succ_{cls} d$.*

At this point we can give the proof that all remaining clauses $c \in C$ are true in I. Assume that for all d, with $c \succ d$, we already know that I makes d true. We show that c is true in I through the following case analysis.

- c is a clause in which a negative atom of form $t \not\approx t$ is selected. Write c in the form $[t \not\approx t] \cup R$. Clearly, equality reflexivity can be applied on c. The result is R. Because C is saturated, either $R \in C$, or there are clauses $r_1, \ldots, r_n \in C$, s.t. $R \succ r_1, \ldots, r_n$ and $r_1, \ldots, r_n \models R$. By induction, $r_1, \ldots, r_n$ are true in I. Then also R must be true in I.

  In either case, R is true in I and this implies that c is true in I.

  (In the rest of this proof, there will be similar arguments, but we will omit the reasoning involving redundancy. We will simply show that a forward inference is possible, and that the conclusion is smaller. We will then simply assume that the conclusion is true)

- c is a clause in which a negative atom of form $t_1 \not\approx t_2$ with $t_1 \neq t_2$ is selected. Write c in the form $[t_1 \not\approx t_2] \cup R$. Assume that $t_1 \succ t_2$. If t_1 is in normal form (under I) then the normal form of t_2 is certainly not t_1, because rewriting makes a term $\succ$-smaller. Hence c is true in I.

  If t_1 is not in normal from under I, then there is a rewrite rule $(u_1 \rightarrow u_2) \in I$ that can rewrite t_1. By Theorem 3.3.10, there is a clause $[u_1 \approx u_2] \cup S$ in C, s.t. $u_1 \approx u_2$ is selected, and S is false in I.

Because $t_1 \not\approx t_2$ and $u_1 \approx u_2$ are selected, $t_1 \succ t_2$, and $u_1 \succ u_2$, negative superposition is possible, which results in some clause d, with form $[t_1' \not\approx t_2] \cup R \cup S$, and for which $t_1 \rightarrow_{\{u_1 \rightarrow u_2\}} t_1'$. Since we know that C is saturated, we know that d is true in C. Since S is false in C, it must be the case that either $t_1' \not\approx t_2$ or R is true in I. If R is true, then clearly c is true in R. If $t_1' \not\approx t_2$ is true, then t_1' and t_2 have distinct normal forms under I. Since $t_1 \rightarrow_I t_1'$ and I is confluent, t_1 has the same normal form as t_1'. Hence $t_1 \not\approx t_2$ is true in I.

- c is a clause in which a positive atom of form $t \approx t$ is selected. Then c is trivially true.
- c is a clause in which the selected atom has form $t_1 \approx t_2$ with $t_1 \succ t_2$, and there is another equality $t_1 \approx t_3$ in c, s.t. J_c makes $t_2 \approx t_3$ true. Then I makes also $t_2 \approx t_3$ true. Write c in the form $[t_1 \approx t_2,\ t_1 \approx t_3] \cup R$. It is possible to apply equality factoring on $[t_1 \approx t_2,\ t_1 \approx t_3] \cup R$, which results in $[t_1 \approx t_3,\ t_2 \not\approx t_3] \cup R$. We may assume by induction that this clause is true in I. Because it is impossible that $I \models t_2 \not\approx t_3$, it must be the case that $[t_1 \approx t_3] \cup R$ is true in I. Hence c is true in I.
- c is a clause in which the selected atom has form $t_1 \approx t_2$ with $t_1 \succ t_2$. Assume that t_1 is not in normal form with respect to J_c. Write c in the form $[t_1 \approx t_2] \cup R$.

  Because t_1 is not normal, it can be rewritten by a rewrite rule $(u_1 \rightarrow u_2) \in J_c$. By Theorem 3.3.10, there is a clause $[u_1 \approx u_2] \cup S$ in C, s.t. $u_1 \approx u_2$ is selected, and S is false in I.

  Because both $t_1 \approx t_2$ and $u_1 \approx u_2$ are selected, $t_1 \succ t_2$ and $u_1 \succ u_2$, positive superposition is possible, and there must be a clause d of form $[t_1' \approx t_2] \cup R \cup S$ in C, for which $t_1 \Rightarrow_{\{u_1 \rightarrow u_2\}} t_1'$. Because d is smaller than c, we may assume that d is true in I. Since we know that S is false in I, it must be the case that either $t_1' \approx t_2$, or R is true in I. If R is true, then clearly c is also true. If $t_1' \approx t_2$ is true in I, then t_1' and t_2 have the same normal form $\hat{t}$. Because $t_1 \Rightarrow_I t_1' \Rightarrow_I^* \hat{t}$, also $t_1 \approx t_2$ must be true in I. We have shown that c is true in I.

We give some examples illustrating aspects of the proof:

Example 3.3.12 *Suppose the saturated set C contains a clause $\{a \approx b\} \cup R$ in which $a \approx b$ is selected, and $a \succ b$. Suppose that $a \rightarrow b$ was added to I.*
Now if C contains also a clause $\{f(a,a) \approx \mathbf{true}\} \cup S$ in which $f(a,a) \approx \mathbf{true}$ is selected, the equation $f(a,a) \rightarrow \mathbf{true}$ will be not added to I, because $f(a,a)$ is not in normal form.
In that case, however, because C is saturated, one of the following clauses must be redundant in C, and hence true in I : (Here we use induction and the definition of redundancy)

$$\{f(a,b) \approx \mathbf{true}\} \cup R \cup S \text{ or } \{f(b,a) \approx \mathbf{true}\} \cup R \cup S.$$

Let's assume it is the first. We know that no literal in R *is true. Hence the clause* $\{f(a,b) \approx \mathbf{true}\} \cup S$ *is true. If an atom in* S *is true, then* $f(a,a) \approx \mathbf{true}\} \cup S$ *is true. Otherwise, it must be the equality* $f(a,b) \approx \mathbf{true}$ *that is true is true. Because* I *contains the rewrite rule* $a \rightarrow b$*, also the atom then* $f(a,b) \approx \mathbf{true}$ *is true.*

Example 3.3.13 *In the previous example, we used the fact that* $a \rightarrow b$ *is added only when* R *is false, so that* $a \approx b$ *is the only true atom in* $\{a \approx b\} \cup R$*.*

However, consider the following clause $\{a \approx b,\ a \approx c\} \cup R'$ *and assume that* I *contains the rule* $b \rightarrow c$*.*

In that case, adding $a \rightarrow b$ *to* I*, will simultaneously make the atom* $a \approx c$ *true, and we will have problems proving that the clause* $\{f(a,a) \approx \mathbf{true}\} \cup S$ *is true.*

It is for this reason that one needs equality factoring. Equality factoring results in the clause $\{a \approx c,\ b \not\approx c\} \cup R'$*. If* $a \approx c$ *is not selected, then it will not be added, and in that case* $f(a,a) \approx \mathbf{true}$ *is in normal form. If* $a \approx c$ *is selected, then it is added, but then the rest* $\{b \not\approx c\} \cup R'$ *is false. In both case, we have no problems proving the truth of* $\{f(a,a) \approx \mathbf{true}\} \cup S$*.*

Example 3.3.14 *Usual factoring is a special case of equality factoring. Consider the clause* $[P \approx \mathbf{true},\ P \approx \mathbf{true}] \cup Q$*. It has an equality factor* $[P \approx \mathbf{true},\ \mathbf{true} \not\approx \mathbf{true}] \cup Q$*. The disequality* $\mathbf{true} \not\approx \mathbf{true}$ *will be removed by equality reflexivity.*

Chapter 4

Propositional SAT-solving

4.1 Introduction

Definition 4.1.1 *We assume a set of propositional symbols $\mathcal{P}$. We will also use the term* atom *for elemens of $\mathcal{P}$. A* literal *is an element p of $\mathcal{P}$ or its negation $\neg p$. Literals of form p are called* positive. *Literals of form $\neg p$ are called* negative.

We assume for each $p \in P$ that $\neg p \neq p$, and $\neg\neg p = p$.
A clause *is a finite set of literals. A* clause set *is a finite set of clauses.*

Definition 4.1.2 *The set of* truth-values *(also* booleans*) equals* $\{\text{false}, \text{true}\}$. *An* interpretation *$I$ is a partial function that assigns truth-values to some elements of $\mathcal{P}$. We say that I is* the empty interpretation *if I does not make any assignments at all. We say that I* completely interpretes *a clause or clauseset c if I is defined for all atoms in c.*
Let I be an interpretation. Let c be a clause. We define $I(c) = \text{true}$ if either c contains a positive literal p for which $I(p) = \text{true}$, or a negative literal $\neg p$ for which $I(p) = \text{false}$.
We define $I(c) = \text{false}$ if for each literal L in c holds: Either L is a positive atom p and $I(p) = \text{false}$, or L is a negative atom $\neg p$ and $I(p) = \text{true}$.
For a clause set C, we define $I(C) = \text{true}$ if for each $c \in C$, it is the case that $I(c) = \text{true}$. We define $I(C) = \text{false}$ if there is a $c \in C$, such that $I(c) = \text{false}$.

Definition 4.1.3 *Let C be a propositional clause set. We say that C is satisfiable if there exists an interpretation I such that $I(C) = \textbf{true}$. Otherwise, C is* unsatisfiable.

4.2 Satisfiability Testing: A first Algorithm

The problem of determining whether a propositional clause set can be true in an interpretation, is NP-complete. As a consequence, all known algorithms are exponential. One possible algorithm for determining satisfiability would be resolution. However, it has turned out that algorithms that directly search for an interpretation are more successful than resolution.

The idea behind the search algorithms is straightforward. Let I is an interpretation and let C be a clause set. We want to know whether I can be extended to an interpretation that makes C true. If there is already a false clause in C, then we can give up immediately. If all clauses in C are true, then we have found a solution. Otherwise, we find an unassigned atom in C, and try to assign it in both possible ways.

Definition 4.2.1 *Let I_1 and I_2 be interpretations. We say that I_2 is* an extension *of I_2 if for all $p \in P$,* $I_1(p) = \text{true}$ *implies* $I_2(p) = \text{true}$ *and* $I_1(p) = \text{false}$ *implies* $I_2(p) = \text{false}$.

The algorithm in this section is based on the following simple properties:

Lemma 4.2.2

1. *Let I be an interpretation. Let C be a clause set. If* $I(C) = \text{false}$, *then in every extension I' of I,* $I'(C) = \text{false}$.
2. *Let I be an intepretation. Let C be a clause set.* $I(C) = \text{true}$ *if-and-only-if there exists an extension I' of I that is defined for all atoms in C, s.t.* $I'(C) = \text{true}$.
3. *Let I be an interpretation. Let p be an atom for which I is undefined. Let I_t be obtained by extending I with the assignment* $I(p) = \text{true}$. *Let I_f be obtained by extending I with the assignment* $I(p) = \text{false}$.

   Let C be a clause set. There exists an extension I' of I for which $I'(C) = \text{true}$ *if-and-only-if there exists an extension I' of I_f in which $I'(C)$ is true or there exists an extension I' of I_t in which $I'(C)$ is true.*

Example 4.2.3 *The following set of clauses is unsatisfiable:*

$\{P_1, Q_1, A\}$,
$\{\neg P_1, Q_1\}$,
$\{P_1, \neg Q_1\}$,
$\{\neg P_1, \neg Q_1\}$,

$\{P_2, Q_2, \neg A\}$,
$\{\neg P_2, Q_2\}$,
$\{P_2, \neg Q_2\}$,
$\{\neg P_2, \neg Q_2\}$.

Algorithm 4.2.4

```
bool sat( interpretation& I, const clauseset& C )
{
   if C contains a clause c, s.t. I(c) = false, then
      return false;
```

```
      if for all clauses c in C,  I(c) = true, then
         return true;

      Let p be an atom for which I(p) is undefined, occurring in C;
      unsigned int m = the number of assigned atoms in I;

      Extend I with the assignment I(p) = false;
      Let bool b = sat(I,C);
      if( b )
         return b;
      Restore I to length m;
      Extend I with the assignment I(p) = true;
      Let bool b = sat(I,C);
      return b;
   }
```

Algorithm 4.2.5

```
   bool sat( interpretation& I, used& U, const clauseset& C )
   {
      if C contains a clause c, s.t. I(c) = false, then
      {
         for all atoms p occurring in c, assign U(p) = true.
         return false;
      }

      Let p be an atom for which I(p) is undefined, occurring
      in C;

      unsigned int m = the number of assignments in I;

      Extend I with the assignment I(p) = false.
      Extend U with the assignment U(p) = false.

      Let bool b = sat(I,C);
      if(b) return b;

      Let bool u = U(p);

      Restore I to length m;
      Resture U to length m;

      if(u) return false;
         // Because conflict did not depend on the assignment to p.

      Extend I with the assignment I(p) = true;
      Extend U with the assignment U(p) = false;
```

```
    return sat(I,U,C);
}
```

4.3 DPLL

While searching for an interpretation, one has to guess truth-assignments to the variables. Because n variables can have 2^n truth assignments, naive search is exponential. Algorithm 4.2.4 is somewhat better than the naive algorithm, because in case it detects a false clause in a partial interpretation, it will not further attempt to extend it into a complete interpretation. We have seen that Algorithm 4.2.5 sometimes performes better than Algorithm 4.2.4 because it both truth-values only for those atoms that contributed to a false clause. However, we have also seen that on most problems, most atoms will be marked, so that the actual saving is somewhat disappointing.

Much more saving can be obtained by trying to postpone guesses as much as possible. Instead when one should try to derive *deterministic consequences* as much as possible. If one considers at example 4.2.3, and one has assigned $I_(P_1) =$ true, then the second clause can be made true only if $I(Q_1) =$ true. Therefore the search algorithm should not wait until it arrives at Q_1, but immediately assign $I(Q_1) =$ true when it assigns $I(P_1) =$ true.
The DPLL-algorithm looks for clauses c that contain one or zero unassigned atoms, and for which all the assigned atoms are assigned 'the wrong way'. In case there is no unassigned atom, the clause is false and we have backtrack. In case, there is one unassigned atom, this atom is our last chance to make the clause true, and we assign it in such a way that the clause becomes true. Since the new assignment may make other clauses 'almost false', the algorithm repeats this process until no further assignments are possible. Only then it selects an atom, and guesses the truth value for this atom.

Lemma 4.3.1 *Let I_1 be an interpretation. Let C be a clause set. Let c be a clause of C containing exactly one unassigned atom p, such that $I_1(c \backslash p) =$ false.*

Let I_2 be obtained by extending I_1 as follows: If the occurrence of p in c is positive, then I_2 is obtained by adding the assignment $I_1(p) =$ true to I_1. If the occurrence of p in c is negative, then I_2 is obtained by adding the assignment $I_1(p) =$ false to I_1.

Then: There exists an extension I' of I_1 for which $I'(C) =$ true iff there exists an extension I' of I_2 for which $I'(C) =$ true.

Algorithm 4.3.2

```
bool dpll( interpretation& I, used& U, parents& P, const clauseset& C )
{

   while( ( C contains a clause c, s.t. I(c) = false ) ||
          ( C contains a clause c which contains an atom p, s.t.
    I(c \ p) = false ) )
```

```
{
   if( C contains a clause c, s.t. I(c) = false )
   {
      for all atoms p occurring in C, assign U(p) = true;
      return false;
   }

   if( C contains a clause c,
       which contains a single unassigned atom p, and
       I( c \ p ) is false )
   {
      if( the occurrence of p in c is positive )
      {
         Extend I with I(p) = true;
         Extend U with U(p) = false;
         Extend P with P(p) = the set ( c \ p );
      }
      else
      {
         // The occurrence must be negative.

         Extend I with I(p) = false;
         Extend U with U(p) = false;
         Extend P with P(p) = the set ( c \ p );
      }
   }
}

if( for all clauses c in C,   I(c) = true ) return true;

Let p be an unassigned propositional symbol occurring in C;

Extend I with I(p) = false;
Extend U with U(p) = false;
Extend P with P(p) = { };

unsigned int m = the number of assigned atoms in I;

bool b = dpll( I, U, P, C );
if(b) return true;

while( U,P,I have length greater than m )
{
   Let q be the last atom assigned in U,P,I.
   If U(q) then for each atom r in P(q), assign U(r) = true;
   Remove q from U;
```

```
         Remove q from P;
         Remove q from I;
      }

      bool u = U(p);
      Remove p from U;
      Remove p from P;
      Remove q from I;

      if( !u ) return false;

      Extend I with I(q) = true;
      Extend U with U(q) = false;
      Extend P with P(p) = { };

      return dpll( I, U, P, C );
   }
```

We demonstrate the effect of Algorithm 4.3.2 on example 4.2.3. Initially, I, P and U are empty. No forward reasoning is possible. Assume that the algorithm chooses P_1 for guessing. Then we have

```
I(P1) = false,   U(P1) = false,   P(P1)  = { },            m = 1.
```

dpll recursively calls itself, it will enter the while-loop with $c = \{P_1, \neg Q_1\}$, and $p = Q_1$. The result will be

```
I(P1) = false,   U(P1) = false,   P(P1) = { }
I(Q1) = false,   U(Q1) = false,   P(Q1) = { P1 }.
```

Next it will enter the while-loop with $c = \{P_1, Q_1, A\}$ and $p = A$. The result is the state:

```
I(P1) = false,   U(P1) = false,   P(P1) = { },             m = 1,
I(Q1) = false,   U(Q1) = false,   P(Q1) = { P1 },
I(A)  = true,    U(A)  = false,   P(A) =  { P1, Q1 }.
```

Now, no deterministic reasoning is possible, and the algorithm needs to guess again. Assume it guesses on P_2. We obtain

```
I(P1) = false,   U(P1) = false,   P(P1) = { },             m = 4,
I(Q1) = false,   U(Q1) = false,   P(Q1) = { P1 },
I(A)  = true,    U(A)  = false,   P(A)  = { P1, Q1 },
I(P2) = false,   U(P2) = false,   P(P2) = { },
```

Note that m is a local variable, and the value $m = 1$ still exists on the top level. After the guessing, dpll calls itself. It will enter the while loop with $c = \{P_2, Q_2, \neg A\}$ and $p = Q_2$. The result is:

```
I(P1) = false,   U(P1) = false,   P(P1) = { },
I(Q1) = false,   U(Q1) = false,   P(Q1) = { P1 },
I(A)  = true,    U(A)  = false,   P(A)  = { P1, Q1 },
I(P2) = false,   U(P2) = false,   P(P2) = { },
I(Q2) = true,    U(Q2) = false,   P(Q2) = { A, P2 },
```

Now $\{P_2, \neg Q_2\}$ is false. After returning, we have

```
I(P1) = false,   U(P1) = false,   P(P1) = { },          m = 4,
I(Q1) = false,   U(Q1) = false,   P(Q1) = { P1 },
I(A)  = true,    U(A)  = false,   P(A)  = { P1, Q1 },
I(P2) = false,   U(P2) = true,    P(P2) = { },
I(Q2) = true,    U(Q2) = true,    P(Q2) = { A, P2 },
```

After returning, there is a while-loop, which will be entered one time. During the loop $U(A)$ will be assigned true. The result is:

```
I(P1) = false,   U(P1) = false,   P(P1) = { },          m = 4,
I(Q1) = false,   U(Q1) = false,   P(Q1) = { P1 },
I(A)  = true,    U(A)  = true,    P(A)  = { P1, Q1 },
I(P2) = false,   U(P2) = true,    P(P2) = { },
```

We describe in words how the algorithm continues. Because $U(P_2)$ = true, it will continue attempting $I(P_2)$ = true. From $I(P_2)$ = true, it will derive $I(Q_2)$ = true through the clause $\{\neg P_2, Q_2\}$. Now $I(\{\neg P_2, \neg Q_2\})$ = false. The algorithm will backtrack all the way down to the assignment $I(P_1)$ = false. It will enter the state

```
I(P1) = true,    U(P1) = false,   P(P1) = { }.
```

It is easily checked that one step of forward reasoning will result in a false clause, after which the algorithm will be complete.

4.4 DPLL with Learning

The idea behind DPLL-with learning is simple, but extremely important. It is important because the idea behind is applicable (and useful) to all backtracking algorithms, not only those for satisfiability testing. In order to understand it, we take a closer look at the relevance marking of Algorithms 4.2.5 and 4.3.2.

The intuitive justification of relevance marking is as follows: If some choice did not contribute to the conflict that caused the failure, then there is no need in considering alternatives of this choice, because the same failure will be reproduced.

Formally, the semantics is as follows: Suppose dpll returns, and did not find an interpretation. Let I' be the interpretation I restricted to the atoms p for which $U(p)$ = true. Then there is no extension of I'' such that $I''(C)$ = true.

For example, suppose that dpll returns in the following situation:

```
I(P) = false,    U(P) = true,    P(P1) = { },          m = 4,
I(Q) = true,     U(Q) = false,   P(Q1) = { P1 },
I(R) = true,     U(R) = true,    P(A)  = { P1, Q1 },
I(S) = false,    U(S) = false,   P(P2) = { }.
```

Then we know that there is no interpretation extending the interpretation I' defined by $I'(Q) = \text{true}$, $I'(S) = \text{false}$. Because of this, it makes no sense to try $I(S) = \text{true}$ without changing the assignment to either of Q, R.

Lemma Learning is obtained when the knowledge that is implicit in the markings U, is made explicit in clauses. We know the interpretation I' with $I'(Q) = \text{true}$, $I'(S) = \text{false}$ cannot be extended to an interpretation that makes all clauses of C true. This knowledge can be encoded as the clause $\{\neg Q, S\}$ and this can be added to C. We call such a clause a *learnt clause.*

Making the relevancy explicit in clauses have several major advantages:

1. Upon backtracking an atom p, all knowledge about the relevancy of p is lost. If we permanently the learnt clause to C, this knowledge will be preserved.

2. Learnt clauses analyse the conflicts much more accurately than relevance markers. Splits that later turn out irrelevant, may be still partially analyzed, and therefore get marked.

3. We will see that all learnt clauses are derivable by resolution. As a consequence, it becomes easy for the dpll-algorithm to output *proofs.*

In order to see an example of (2), consider the clause set $C_1 \cup C_2$. Suppose, the algorithm splits on p. It may be the case $I(p) = \text{false}$ cannot be extended into an interpretation for which $I(C_1) = \text{true}$, but that C_2 alone is unsatisfiable. Then the assumption p may be marked as relevant. When later the algorithm 'discovers' that C_2 is alone unsatisfiable, the marks will be not undone.

Lemma 4.4.1 *Let I_1 be an interpretation. Let c be a clause of C containing exactly one unassigned atom p, such that $I_1(c\backslash p) = \text{false}$.*

Let I_2 be obtained by extending I_1 as follows: If the occurrence of p in c is positive, then I_2 is obtained by adding the assignment $I_1(p) = \text{true}$ to I_1. If the occurrence of p in c is negative, then I_2 is obtained by adding the assignment $I_1(p) = \text{false}$ to I_1. Let d be a clause for which $I_2(d) = \text{false}$. Then either:

1. *$I_1(d) = \text{false}$, or*

2. *c and d can resolve on atom p, and for the result e holds: $I_1(e) = \text{false}$.*

Proof
If d is false in I_1, then the proof is obviously complete. Otherwise, observe that the only difference between I_1 and I_2 is in the assignment to p. We may assume that $I_2(p) = \text{true}$, because the situation where $I_2(p) = \text{false}$ is completely analogeous.

Because $I_2(d) = \text{false}$, it is possible to write $d = \{\neg p\} \cup d'$, where $I_2(d') = \text{false}$, and p does not occur in d', not positively and not negatively.
Because the original clause c contains p positively, we can resolve c and d on atom p. The result equals $(c \backslash p) \cup d'$. By assumption, we have $I_1(c \backslash p) = \text{false}$. We have just seen that $I_1(d') = \text{false}$. It follows that $I_1((c \backslash p) \cup d') = \text{false}$.

Algorithm 4.4.2

```
union< clause, interpretation >
sat( interpretation& I, parents& P, clauseset& C )
{
   while( C contains a clause c, s.t. either I(c) = false,
          or I(c) is not known to be true and exactly one atom
          in c is not assigned by I )
   {
      if( C contains a clause c, for which I(c) = false )
         return c;

      if( C contains a clause c, which contains a single unassigned
          atom p, and I( c \ p ) = false )
      {
         if( the occurrence of p in c is positive )
         {
            Extend I with I(p) = true;
            Extend P with P(p) = c;
         }
         else
         {
            // The occurrence of p in c must be negative.

            Extend I with I(p) = false;
            Extend P with P(p) = c;
         }
      }
   }

   if( for all clauses c in C,   I(c) = true )
      return I;

   Let p be a symbol occurring in C that is unassigned by I;

   Extend I with I(p) = false;
   Extend P with P(p) = some nonsense clause.

   unsigned int m = the number of assigned atoms in I;
```

```
   union< clause, interpretation > u = sat( I, P, C );

   if( u is an interpretation )
      return u;

   Let c be the clause in u;

   while( I, P have length greater than m )
   {
      Let q be the last atom that is assigned in I, P.

      if( ( -q ) occurs in c )
         c = the resolvent of c and P(q) using atom q.

      Remove q from I;
      Remove q from P;
   }

   Remove p from I;
   Remove p from P;

   if( the atom p does not occur in c )
   {
      // This means that the conflict was not caused by assuming
      // I(p) = false.

      return c;
   }

   Insert c into C;

   // sat will immediately use c for forward reasoning, so that
   // I(p) = false will be assigned.

   return sat( I, P, C );
}
```

Bibliography

[CL73] C.-L. Chang and R.C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[Gal86] Jean H. Gallier. *Logic for Computer Science (Foundations of Automatic Theorem Proving)*. Harper and Row Publishers, 1986.

[Lei88] A. Leitsch. On some formal problems in resolution theorem proving. In *Yearbook of the Kurt Gödel Society*, pages 35–52, 1988.

[Lov78] D. W. Loveland. *Automated Theorem Proving, a logical basis*. North-Holland Publishing Compagny, 1978.

[Min88] G. Mints. Gentzen-type systems and resolution rules, part 1, propositional logic. In *COLOG-88, International Conference on Computational Logic*, 1988.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.