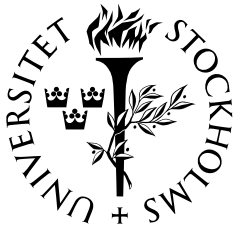

SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Automated Theorem Proving

av

Tom Everitt

2010 - No 8

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Automated Theorem Proving

Tom Everitt

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Rikard Bøgvad

2010

Abstract

The calculator was a great invention for the mathematician. No longer was it necessary to spend the main part of the time doing tedious but trivial arithmetic computations. A machine could do it both faster and more accurately. A similar revolution might be just around the corner for proof searching, the perhaps most time consuming part of the modern mathematician's work. In this essay we present the Resolution procedure, an algorithm that finds proofs for statements in propositional and first-order logic. This means that any true statement (expressible in either of these logics), in principle can be proven by a computer. In fact, there are already practically usable implementations available; here we will illustrate the usage of one such program, Prover9, by some examples. Just as many other theorem provers, Prover9 is able to prove many non-trivial true statements surprisingly fast.

Contents

1 Introduction

An automated theorem prover is an algorithm that uses a set of logical deduction rules to prove or verify the validity of a given proposition. So if logical deductions are perceived of as a formalization of correct, sound steps of reasoning, then an automated theorem prover becomes an *algorithm of reasoning*. As such, the study of automated theorem proving forms a quite natural extension to the mathematical-philosophical programme of formalizing thought (i.e., logic) initiated by Aristotle 300BC.

1.1 Theoretical Preliminaries

Sentences of almost any logic can be divided into three groups: the valids, the contradictorious and the ones in between. The difference is their semantical status. The valid ones are tautologies, they are true no matter how interpreted; the contradictorious are unsatisfiable, they can never be true. The rest are sentences which are sometimes true, but not always.

The set of valid sentences $L_\top$ is almost the same as the set of contradictorious sentences $L_\bot$, the slight difference being the addition of a negation sign. If P is a valid sentence, $\neg P$ will be contradictorious, and vice versa.

In propositional logic there are never more than a finite number of interpretations (valuations), making the partitioning of a propositional language L into $L_\top$, $L_\bot$ and $L_{rest} = L - (L_\top \cup L_\bot)$ theoretically easy. To decide which set a given sentence belongs to, we can simply test all possible valuations. The problem of doing this in practice is less easy, and will be discussed later.

In predicate logic the situation is more complex. The number of interpretations are uncountably infinite, making any "brute force"-procedure impossible. We depend entirely on more profound techniques to determine semantical status. Sometimes convincing informal arguments are used, e.g. to motivate axioms. But most often we turn to *proofs*.

Proofs are sequences of sentences $S_1, \ldots, S_n$, where each sentence S_i is either an axiom (whose validity is originally established by an informal argument), or follows from sentences occurring earlier in the sequence by some *deduction rule* (i.e. a relation $P, Q \vdash R$ between formulas P, Q, R, such that whenever P and Q are true under an interpretation I, then so are R). Since the deduction rules are chosen to preserve validity, the validity of S_n is proven.

Proofs can be used in two ways. Either, as above, to prove the validity of S_n from axioms; sometimes called *forward-proofs*. But they can also be used to derive the unsatisfiability of a formula Q, by proving the negation of an axiom from Q (a *backward-proof*, aka proof by contradiction). The latter technique turns out in practice to be better suited for automatic proof procedures, because no matter what the original formula is, the goal is always the same (the negation of some axiom).

In the resolution method, for instance, the goal is the empty clause (which is contradictorious). Consequently, no matter the input, it always tries to derive a clause with all literals eliminated.

Completeness theorems for the respective logics ensures that a sentence is valid if and *only if* a proof is available (given some suitable set of deduction rules). Furthermore, the proofs are only countably many: they are strings of symbols, and it is easy to determine whether a string of symbols is a proper proof or not.

Hence proofs provide us with a way to *verify* (in finite time) that a given sentence P is valid (or contradictorious). Systematically stepping through an enumeration of all proofs until a proof of P is found, is a distinct theoretical possibility (although hardly a practical or efficient one).

Having thus in principle found a way to verify that a sentence belongs to $L_\top$ or $L_\bot$, we would expect to find a way to verify that a sentence R is in L_{rest}. After all, this problem seems much easier. We do not need to show anything about *all* interpretations, we only need to find *one* interpretation in which R is true, and *one* under which R is false.

Having also a method to verify satisfiability, we would always be able to determine the belonging of a given sentence S. We could run the validity and satisfiability procedures simultaneously. One of them would always stop in finite time.

Although it is possible in many situations to find two interpretations that gives different truth values to a formula R, it is not, surprisingly, possible to find an algorithm that (always) succeeds. This is a consequence of a result proved independently and almost simultaneously by Alonzo Church and Alan Turing, in the 1930's.

Theorem 1.1 (Turing's Theorem)**.** *There exists no algorithm A such that A always finishes with the correct answer (in finite time) to the question: is the formula P a valid sentence in first-order predicate logic?*

A somewhat analogous result for propositional logic essentially states that there is no *efficient* procedure to determine the semantical status of a propositional formula. The satisfiability problem for propositional problem is NP-complete[1]. This makes most scientists believe that no algorithm A exists such that (i) A always answers correctly to the question: is the propositional formula P satisfiable? and (ii) A always terminates in polynomial time with respect to the size of the input. (NP-completeness has not yet been *proven* to imply that no polynomial time algorithm exists, however.)

Theorem 1.2 (Cook's Theorem)**.** *The satisfiability problem of propositional logic is NP-complete.*

1.2 About This Essay

In this essay automated theorem proving in propositional and first-order logic will be covered. The main part of the material comes from essentially two sources. The first is Alan Robinson's paper "A Machine-Oriented Logic Based on the Resolution Principle"[5] from 1965. This is the paper where the resolution proof procedure was first presented. The resolution procedure has ever since dominated the area of automated theorem proving in first-order logic (cf. [3, p. ix]). Accordingly it dominates also in this essay.

The second source is a textbook by Melvin Fitting, named "First-Order Logic and Automated Theorem Proving"[3], which has provided me the more general picture of automated theorem proving.

I claim no originality of the ideas presented in the subsequent parts of this essay. Although I have always written from my own understanding, the original proof ideas etc. comes almost exclusively from the two sources I have mentioned.

On the reader's part, I have assumed that he/she possesses a basic knowledge of formal logic, approximately corresponding to a first course in logic. So familiarity with: syntax and semantics; proof and validity; completeness, soundness and compactness is presumed, but the reader should do fine without any prior acquaintance with automated theorem proving. Some important important logical concepts will be reviewed, however.

2 Propositional Logic

We have already sketched a proof procedure for sentences of propositional logic in the introduction. There we used the simplest method at hand: we tried all possible valuations of the occurring propositional letters.

In this section we will develop an alternative method called Propositional Resolution. It is based on a the (propositional) resolution rule, a simple variant of the first-order procedure that we will use later.

Its main virtue is that it forms a natural introduction to the Predicate Resolution procedure, but it also shares some similarities with one of the most efficient proof procedures of propositional logic: The Davis-Putnam procedure for propositional logic[2]. The Davis-Putnam procedure for propositional logic will not be discussed in this essay however.

The Resolution Rule. In logic we generally motivate our choice of deduction rules by their simpleness and intuitiveness. A good example of this is the Modus Ponens rule:

$$A, (A \to B) \vdash B$$

which could hardly be any simpler or more intuitive.

In the case of automated theorem proving we are more interested in the *speed* of the deductions, or, rather, deduction rules upon which we can build fast proof procedures. It turns out that the *resolution rule* is suitable:

$$(A_1 \vee \cdots \vee A_k \vee C), (B_1 \vee \cdots \vee B_l \vee \neg C) \vdash (A_1 \vee \cdots \vee A_k \vee B_1 \vee \cdots \vee \ B_l)$$

The key point is that C occurs in one disjunction and $\neg C$ in another. Hence we may eliminate C. Because any model satisfying the disjunctions to the left must *fail* to satisfy either C or $\neg C$. In case it fails to satisfy C it must satisfy at least one of $A_1 \ldots, A_k$. And in case it fails to satisfy $\neg C$, it must satisfy at least one of $B_1, \ldots, B_l$.

In either case the model satisfies at least one of $A_1, \ldots, A_k, B_1, \ldots, B_l$, and thereby the right hand formula, the *resolvent*, $(A_1 \vee \cdots \vee A_k \vee B_1 \vee \cdots \vee \ B_l)$.

Refutation Procedures. Based on the resolution rule we can build a procedure that tells whether a given sentence is unsatisfiable or not. If we can deduce a contradiction, it must be unsatisfiable, if we cannot, it must be satisfiable.

This can also be used to determine validity. Say for example that we want to know whether a formula P is valid or not. The trick is then to let the procedure determine the satisfiability of $\neg P$. If $\neg P$ was satisfiable, then P is not valid, and if $\neg P$ was not satisfiable, then P is valid.

A procedure determining validity of P by refuting $\neg P$ (i.e. proving that $\neg P$ is contradictorious), is called a *refutation* procedure. Most ATP:s (including Resolution) are refutation procedures.

Before further investigating the resolution procedure, we need to settle some terminology, as well as have a look at some normal forms.

2.1 Definitions

Language Our language consists of *propositional letters* A, B, C,..., indexed with a natural number if necessary; together with the *primary connectives* $\neg$, $\wedge$, $\vee$ and the parentheses '(', ')'. The syntactic rules for their combination are the standard ones. We call a well formed string of these symbols a *formula* or *sentence*.

Other connectives such as $\rightarrow$, $\leftrightarrow$ are not considered primary. They are defined as $(A \rightarrow B) = ((\neg A) \vee B)$ and $(A \leftrightarrow B) = (A \rightarrow B) \wedge (B \rightarrow A)$.

Literal. With a *literal* we mean a formula that is a propositional letter or the negation of a propositional letter. A, $\neg C$ and $\neg B_1$ are all examples of literals, in contrast to $(A \vee B)$ and $\neg\neg B$ that are non-literals.

Complement. For a propositional letter A, the *complement* of A is $\overline{A} = \neg A$, and the complement of $\neg A$ is $\overline{\neg A} = A$. So the complement of a literal is also a literal. Two literals form a *complementary pair* if they are each others complements.

The reason for introducing complements is that it is a convenient way of avoiding double negations. The complement of the complement of A, for example, is simply A; whereas the negation of the negation of A is $\neg\neg A$.

Valuation. It is well-known that a valuation V of a formula P, is determined by a valuation of the propositional letters in P.

We will use this by identifying a valuation with a set V of literals, such that

- for every propositional letter A in P, either A or $\neg A$ is in V, and
- V contains no complementary pairs.

For a propositional letter A in P, we gather that

- V evaluates A to true if A is in V, and
- V evaluates A to false if $\neg A$ is in V.

It is clear from the definition of valuations that exactly one of these cases must arise. The evaluation of the formula P as a whole is then recursively determined from the valuation of its subformulas.

Satisfaction. A valuation *satisfies* a formula P if it evaluates P to true. And it satisfies a set S of formulas if it evaluates each member of S to true.

A formula P is said to be *satisfiable* if there exist some valuation under which P is true, and P is said to be valid if all valuations renders P true.

Two formulas P and Q are *equivalent* if they are satisfied by exactly the same valuations. And P and Q are considered *equisatisfiable* if either both are satisfiable, or both are unsatisfiable.

Generalized conjunctions and disjunctions. Normal forms based on generalized conjunctions and generalized disjunctions will be an important component of much of the subsequent theory. Therefore we will take the time to define some extra operations and notation for these. A formula on the form $(P_1 \wedge \cdots \wedge P_n)$ is a *generalized conjunction*, and a formula on the form $(P_1 \vee \cdots \vee P_n)$ is a *generalized disjunction*, given that they do not contain the same formula more than once (the P_i's must be pairwise distinct).

To emphasize that a formula is a generalized conjunction (disjunction), rather than just any conjunction (disjunction), we will use different parentheses: $\langle P_1 \wedge \cdots \wedge P_n \rangle$ for generalized conjunctions and $[P_1 \vee \cdots \vee P_n]$ for generalized disjunctions.

We shall also apply the following set-conventions to generalized conjunctions and disjunctions:

- The order of the subformulas will be immaterial, e.g. $\langle A \wedge B \rangle$ will be considered the same as $\langle B \wedge A \rangle$.
- The set operations $\cup, -, \in$ receive the following meaning: for $P = \langle P_1 \wedge \cdots \wedge P_m \rangle$ and $Q = \langle Q_1 \wedge \cdots \wedge Q_n \rangle$:
  - $P \cup Q = \langle P_1 \wedge \cdots \wedge P_m \wedge Q_1 \wedge \cdots \wedge Q_n \rangle$ (duplicates removed).
  - If $A = P_i$ for some i, we have that $A \in P$. Otherwise $A \notin P$.
  - $P - \langle P_i \rangle = \langle P_1 \wedge \cdots \wedge P_{i-1} \wedge P_{i+1} \wedge \cdots \wedge P_m \rangle$, also denoted $P - P_i$ when there is no risk of confusion. If $A \notin P$, then $P - \langle A \rangle = P$.

  The same applies to generalized disjunctions. The union of a generalized conjunction and a generalized disjunction is undefined.

The empty generalized conjunction, denoted $\langle\rangle$, is always true; and the empty generalized disjunction, denoted $[]$, always false.

Clause and Dual Clause. A clause is simply a generalized disjunction of literals, and a dual clause the conjunctive counterpart. So if A_1, A_2, A_3 are distinct propositional letters, then $[A_1 \vee \neg A_2 \vee \neg A_3]$ is a clause and $\langle A_1 \vee A_2 \rangle$ a dual clause.

Clauses are a central component of the normal form we will work with the most: the clause normal form (see below).

2.2 Normal Forms

Normal forms fixate the structure of a sentence in some way. This is a very useful feature. In the case of an automated theorem prover, knowing that a sentence is in some normal form drastically reduces the number of cases one needs to take into account.

Generally it is possible to convert any formula P to an equivalent formula Q on some normal form N, by successively rewriting P according to some tautologies. Sometimes, however, the conversion can be performed faster if we only require Q to be *equisatisfiable*. This is for example the case with clause normal form, as we shall see below.

In many cases equisatisfiability is sufficient. To show, for instance, that a formula P is unsatisfiable, it is enough to show that a formula Q that is equisatisfiable to P, is contradictorious.

Three normal forms for propositional logic will be discussed: negation normal form, clause normal form, and dual clause normal form.

2.2.1 Negation Normal Form

A formula is said to be on *negation normal form* if all negation signs prefixes propositional letters.

Conversion. To convert a formula to negation normal form we make use of De Morgans Laws to make negations "travel inwards", and The Law of Double negation to make double negations disappear.

Example. The formula $\neg((P \vee Q) \wedge (\neg P))$ will be converted the following way:

1. $\neg((P \vee Q) \wedge (\neg P))$ (Original formula)
2. $(\neg(P \vee Q) \vee (\neg(\neg P)))$ (De Morgan),
3. $(\neg(P \vee Q) \vee P)$ (Double Negation),
4. $((\neg P \wedge \neg Q)) \vee P)$ (De Morgan).

More formally, to convert a formula P to negation normal form, do the following.

While P is not a literal, depending on the following cases, do:

- If $P = \neg\neg Q$, then replace P with Q, and continue converting Q to negation normal form.
- If $P = \neg(Q_1 \wedge Q_2)$, then replace P with $((\neg Q_1) \vee (\neg Q_2))$ and convert $\neg Q_1$ and $\neg Q_2$ to negation normal form.
- If $P = \neg(Q_1 \vee Q_2)$, then replace P with $((\neg Q_1) \wedge (\neg Q_2))$ and convert $\neg Q_1$ and $\neg Q_2$ to negation normal form.
- If $P = (Q_1 \wedge Q_2)$, then convert Q_1 and Q_2 to negation normal form.
- If $P = (Q_1 \vee Q_2)$, then convert Q_1 and Q_2 to negation normal form.

When this algorithm has been carried out till the end our formula will consist of literals combined only by $\wedge$ and $\vee$ ($\neg$ will only prefix propositional letters).

We verify this with structural induction:

It is clear that the algorithm applied to an atomic formulas will yield an equivalent formula where $\neg$ only prefixes propositional letters (it will simply do nothing at all).

Assume now that it will work on formulas Q_1 and Q_2. Then it is clear that it will also work on the formulas $\neg\neg Q_1$, $\neg(Q_1 \wedge Q_2)$, $\neg(Q_1 \vee Q_2)$, $(Q_1 \wedge Q_2)$ and $(Q_1 \vee Q_2)$.

Hence it will work on any formula.

2.2.2 Dual Clause Normal Form

Dual clause normal form (or sometimes *disjunctive normal form* or simply *clause form*) is quite easy to characterize. A formula on dual clause form is simply a generalized disjunction of dual clauses. So any formula on the form

$$[\langle L_{11} \wedge \cdots \wedge L_{1k_1} \rangle \vee \cdots \vee \langle L_{n1} \wedge \cdots \wedge L_{nk_n} \rangle]$$

is on dual clause form, given that each L_{ij} is a literal.

Conversion. A given sentence S has a finite number of satisfying valuations $V_1, \ldots, V_n$. These can easily be determined by trying all possible valuations for S. And from each valuation V_i we can form the generalized conjunction P_i of all members of V_i. Since all members of V_i are literals, P_i will be a dual clause.

Now, to get an equivalent sentence S' on dual clause form, simply let S' be the generalized disjunction of all P_i:s, i.e. let $S' = [P_1 \vee \cdots \vee P_n]$

S and S' will then be satisfied by exactly the same valuations, and S' be on normal form.

As an example, consider the sentence $R = (A \wedge (B \vee A))$. Two valuations satisfies R, $\{A, B\}$ and $\{A, \neg B\}$. So an equivalent sentence on dual clause form is $R' = [\langle A \wedge B \rangle \vee \langle A \wedge \neg B \rangle]$.

2.2.3 Clause Normal Form

There is also a normal form based on the clause, the *clause normal form* (also known as *conjunctive normal form* or just *clause form*). It is just like the dual clause form, except that the generalized disjunctions switched place with the generalized conjunction. Hence, any formula on the form

$$\langle [L_{11} \vee \cdots \vee L_{1k_1}] \wedge \cdots \wedge [L_{n1} \vee \cdots \vee L_{nk_n}] \rangle$$

where each L_{ij} is a literal, is on clause normal form.

Conversion. To convert a formula S to an equivalent formula S' on clause normal form, let first T be $\neg S$ converted to dual clause normal form. Then let S' be $\neg T$ converted to negation normal form. Then S' will be equivalent to S, and, due to a De Morgan law, S' will be on clause normal form.

Alternative (more efficient) Conversion. In many situations it is sufficient to find an equisatisfiable formula on clause form. This can be done much faster. To convert for example the formula $P = \neg(A \wedge (B \vee C))$, we assign a new propositional letter $\alpha_1, \alpha_2, \ldots$ to every subformula of P. To $(B \vee C)$ we assign α_1, to $(A \wedge (B \vee C))$ we assign α_2 and to $\neg(A \wedge (B \vee C))$ we assign α_3.

Now, we can express the fact that α_1 should be true if and only if at least one of B and C is true, by the clauses

$$[\neg\alpha_1 \vee B \vee C], [\neg B \vee \alpha_1], [\neg C \vee \alpha_1]$$

And to express that α_2 should be true if and only if both α_1 and A are true, we use the clauses

$$[\neg A \vee \neg\alpha_1 \vee \alpha_2], [\neg\alpha_2 \vee A], [\neg\alpha_2 \vee \alpha_1]$$

Finally, the clauses

$$[\neg\alpha_3 \vee \neg\alpha_2], [\alpha_2 \vee \alpha_3]$$

express α_3's relation to α_2.

Now, let Q be the conjunction of all these clauses and $[\alpha_3]$. If Q is satisfied by some valuation V, then V also satisfies P. And, conversely, if V satisfies P, then there is an extension of V that also satisfies Q. Hence we have found an equisatisfiable formula

$$\begin{aligned} Q = \langle & [\neg\alpha_1 \vee B \vee C] \wedge [\neg B \vee \alpha_1] \wedge [\neg C \vee \alpha_1] \wedge \\ & [\neg A \vee \neg\alpha_1 \vee \alpha_2] \wedge [\neg\alpha_2 \vee A] \wedge [\neg\alpha_2 \vee \alpha_1] \wedge \\ & [\neg\alpha_3 \vee \neg\alpha_2] \wedge [\alpha_2 \vee \alpha_3] \wedge \\ & [\alpha_3] \rangle \end{aligned}$$

The example covers virtually any case that can arise. For a general formula we go ahead much the same way. First we assign new predicate letters to each subformula. Then, for each new predicate letter, we express its relation to its subformula by a few clauses. Finally we create a clause that states that the full formula is true, and form the conjunction of them all.

2.2.4 Normal Form

Normal form will just be short for clause normal form in the case of propositional logic.

2.2.5 A Note on Time Complexity

Definition. An algorithm is considered *efficient* if the time it requires only grows polynomially with the size of the input. It is considered *inefficient* if the time grows faster than polynomially, e.g. exponentially. A problem is *easy* if there exists an efficient algorithm solving it.

Conversion to negation normal form is efficient. Essentially only one operation is applied to each subformula, and the number of subformulas in a formula P only grows linearly with the size of P.

The conversions to dual clause and clause form are inefficient, for the number of valuations of P grows exponentially with the number of distinct propositional letters in P (and in the worst case, all propositional letters are distinct). In fact, it could not be that there was an efficient algorithm for conversion to dual clause form. For if there were an efficient procedure, we could use that the satisfiability problem for dual clause formulas is easy (see next section 2.2.6), to get an efficient procedure solving the general satisfiability problem of propositional logic. But due to Cook's theorem 1.2, no such procedure can exist.[1]

The alternative conversion procedure for clause form is efficient however, it grows linearly with the number of subformulas (just as the negation form procedure). The price is, of course, equivalency.

[1] Is *likely* to exist, if one should be precise. See section 1.1.

2.2.6 Proof Procedural Properties

When one wants to determine satisfiability of a formula it is generally advantageous to know that the formula is on some normal form, rather than having no structural information at all. But whether it is on clause or dual clause normal form also has a significant impact on the time it will take to determine satisfiability.

The Dual Clause Form. The dual clause form is especially well suited for determining satisfiability. Take a sentence

$$P = [C_1 \vee \cdots \vee C_n] = [\langle L_{11} \wedge \cdots \wedge L_{1k_1} \rangle \vee \cdots \vee \langle L_{n1} \wedge \cdots \wedge L_{nk_n} \rangle]$$

(the L_{ij}:s literals and the C_i:s dual clauses).

P is then satisfiable if and only if at least one dual clause C_i is satisfiable. But it is easy to see that a conjunction C_i of literals is satisfiable if and only if it contains no complementary pair (i.e. B and $\neg B$ for some propositional letter B).

That means that P is unsatisfiable if and only if each C_i contains a complementary pair. This can be verified in polynomial time in the number of propositional letters and the size of the formula.

The Clause Form. There is no similarly straightforward procedure for formulas on clause form. The problem of determining satisfiability for a formula on clause form is NP-complete; an immediate consequence of Cook's theorem 1.2, and the fact that we efficiently can convert any formula to clause form with maintained satisfiability.

Unfortunately, the clause form is more common than the dual clause form. Not only is it much faster to convert a formula to clause normal form, but it also arises naturally from the predicate logic proof procedures we shall discuss later. One can of course convert a clause form formula to a dual clause form formula, but it is not a practically useful: the time required for conversion is as great as the time required to check satisfiability of a clause form formula directly.

This makes it much more interesting to develop proof procedures for clause form formulas than for dual clause ones. Consequently, both the resolution procedure and the Davis-Putnam procedure (mentioned in the introduction of this section), are developed for clause form formulas. The satisfiability problem for formulas on clause form is generally referred to as the clause normal form satisfiability problem (or simply CNF-Sat) in the literature.

2.3 Propositional Resolution

We are now ready for the proof procedure of this section. It is essentially a *satisfiability checker* for normal form formulas, but as we have seen in the introduction we can easily reduce the proof problem of any formula to the satisfiability problem of a normal form formula by negating the formula and converting it to normal form. Also, the satisfiability problem of a normal form formula will arise naturally by itself in predicate logic that we will come to later.

Example. We introduce a main example that will stay with us through the section, hopefully making the formal definitions more intelligible.

Assume we want a procedure to tell us whether the normal form formula

$$\langle [\neg A \vee B] \wedge [A \vee B] \wedge [\neg B] \rangle$$

is satisfiable or not (we can easily see that it is not, but we want a *procedure* to tell us that).

A natural way to go ahead would be to try to prove a contradiction from the formula. In case it implies a contradiction, it must be unsatisfiable; and in case not, we should be able to find a model for the formula.

Our satisfiability checker will be based on the resolution rule (stated above on page 6) as its only rule of deduction. It will be convenient to restate the resolution rule specifically for clauses, using the extra terminology introduced with generalized conjunctions.

The Propositional Resolution Rule. Assume $P = [A_1 \vee \cdots \vee A_m \vee C]$ and $Q = [B_1 \vee \cdots \vee B_n \vee \neg C]$ are both clauses. Then

$$R = (P - [C]) \cup (Q - [\neg C]) = [A_1 \vee \cdots \vee A_m \vee B_1 \vee \cdots \vee B_n]$$

is the *resolvent* of (the ordered pair) P and Q *with respect to* C. Often we will simply talk about *a resolvent* of P and Q, without specifying the predicate letter C. Then the resolvent will not necessarily be P and Q's only resolvent, as they may have resolvents with respect to other predicate letters as well.

Two properties of this rule will be important:

(i) Any model satisfying P and Q will also satisfy R (this is just to say that the rule is valid, which we argued for above on page 6).

(ii) If P and Q are clauses, then any resolvent R of P and Q must also be a clause. This must be, since R inevitably will be generalized disjunction of only literals if P and Q contain only literals.

Definition. Two sentences S and T are *equisatisfiable* if each model satisfying S satisfies T and vice versa.

Proposition 2.1. *Assume that S is a normal form formula containing clauses P and Q, and that R is a resolvent of P and Q. Then $S \cup \langle R \rangle$ is equisatisfiable with S, and $S \cup \langle R \rangle$ is on clause normal form.*

Proof. First note that any model satisfying all clauses of $S \cup \langle R \rangle$ must also satisfy the clauses of S (they form a subset of the clauses of $S \cup \langle R \rangle$).

Now, assume that a model M satisfies S. Then M satisfies P and Q, and thereby also their resolvent R. Hence M satisfies all clauses of $S \cup \langle R \rangle$.

Finally we note that since R is a clause, $S \cup \langle R \rangle$ will be on normal form. □

Example. Using proposition 2.1 we could then go ahead and *appending* every resolvent we can find to $\langle [\neg A \vee B] \wedge [A \vee B] \wedge [\neg B] \rangle$. Writing the generalized conjunction as a list, we would get:

1. $[\neg A \vee B]$
2. $[A \vee B]$

3. $[\neg B]$
4. $[B]$
5. $[]$

where 4 is a resolvent of 1 and 2, and 5 is a resolvent of 3 and 4. Since 5 is the empty clause, and hence unsatisfiable, we have reached the desired contradiction here.

The success of the above example might tempt us to formulate a procedure for an arbitrary formula S, along the lines of: Take a pair of clauses C and D of S. If they have a resolvent B, update S by substituting B for C and D. Then apply the same procedure on S again.

This would have worked fine in the above example. It would also have had the nice feature of successively reducing the number of clauses; leading to a very efficient proof procedure, had it always worked.

Unfortunately we can only *append* new resolvents, not *substitute* them for old clauses. Otherwise this might have happened:

1. $[\neg A \vee B]$
2. $[A \vee B]$
3. $[\neg B]$
4. $[A]$

(4 is a resolvent of 2 and 3). If we had replaced 2 and 3 with 4, we would have been stuck.

Hence, we can merely append clauses, not replace. We express this formally with the $\mathcal{R}$ operation, which extends a formula with all available resolvents.

2.3.1 The $\mathcal{R}$ Operation.

Definition. For S on normal form and R a clause, we call the formation of $S \cup \langle R \rangle$ from S to *append (the clause) R to S*.

The result of appending to a sentence S each resolvent, of every pair of clauses of S, will be denoted $\mathcal{R}(S)$.

In our main example, where

$$S = \langle [\neg A \vee B] \wedge [A \vee B] \wedge [\neg B] \rangle$$

this means to append the clauses: $[B]$ (from clause 1 and 2), $[\neg A]$ (from clause 1 and 3) and $[A]$ (from clause 2 and 3). Which gives

$$\mathcal{R}(S) = \langle [\neg A \vee B] \wedge [A \vee B] \wedge [\neg B] \wedge [B] \wedge [\neg A] \wedge [A] \rangle$$

To get the empty clause we have to apply the $\mathcal{R}$ operation once more. It is a resolvent of clause 3 and 4 (and of clause 5 and 6!) of $\mathcal{R}(S)$.

Applying $\mathcal{R}$ n times to a formula S will be denoted $\mathcal{R}^n(S)$.

Proposition 2.2. *For any normal form formula S and any natural number n, then $\mathcal{R}^n(S)$ is equisatisfiable with S.*

Proof. It follows directly from proposition 2.1 that $\mathcal{R}(S)$ will be equisatisfiable with S for any normal form formula S. And since $\mathcal{R}$ preserves the sentence on normal form (also by proposition 2.1), applying $\mathcal{R}$ any number of times, will still yield an equisatisfiable sentence. □

Definition. A formula S to which no new clauses are added by the $\mathcal{R}$ operation (i.e. $\mathcal{R}(S) = S$), is said to be *$\mathcal{R}$-satisfied*.

2.3.2 The Procedure

The propositional resolution procedure can now be defined by the pseudocode of Algorithm 1. For a given sentence S (that should be on normal form), it will keep applying the $\mathcal{R}$ procedure until either no new clauses are added (in which case S is satisfiable) or the empty clause is added (S is unsatisfiable).

Algorithm 1 The-Propositional-Resolution-Procedure(S)

```
i ← 0
loop
  i++
  if R^i(S) = R^(i−1)(S) then
    return satisfiable
  else if R^i(S) contains the empty clause then
    return unsatisfiable
  end if
end loop
```

If n is the number of laps we go trough the loop, either of these has to happen before $n \geq 2^{2l}$ (where l is the number of distinct propositional letters of S). For in each application of $\mathcal{R}$ at least one new clause has to be appended (otherwise the first break criteria will be met). But only a finite number of clauses can be constructed from the finite number of propositional letters of S. In fact, exactly 2^{2l} different clauses can be constructed. $2l$ is the number of literals constructable from l letters, and each literal can either be part of, or not be part of, a clause (a clause is completely determined by its literals).

Now, it follows immediately from proposition 2.2 that if the second break criteria is met at some i, i.e. $[] \in \mathcal{R}^i(S)$, then S cannot be satisfiable. But are all $\mathcal{R}$-satisfied formulas not containing [] satisfiable? The following lemma answers that question in the positive.

Definition. A set of literals, not containing any complementary pair, is called a *partial valuation*. A partial valuation V contradicts a clause C if, for each literal L in C, the complement $\overline{L}$ is in V.

A partial valuation V is a *full valuation* (or simply a *valuation*) of a formula S, if for each literal L in S, either L or $\overline{L}$ is in V.

The partial valuation is used to successively build up a full valuation.

Lemma 2.3. *Any $\mathcal{R}$-satisfied formula not containing [] is satisfiable.*

Proof. Assume that S is an $\mathcal{R}$-satisfied formula, $[] \notin S$, containing propositional letters $A_1, \ldots, A_n$. Then we construct a model for S by successively extending

a partial valuation the following way:

Let $M_0 = \emptyset$.
For i between 1 and n, let

$$M_i = \begin{cases} M_{i-1} \cup \{A_i\} & \text{if } M_{i-1} \cup \{A_i\} \text{ does not contradict any clause of } S. \\ M_{i-1} \cup \{\neg A_i\} & \text{if } M_{i-1} \cup \{\neg A_i\} \text{ does not contradict any clause of } S. \end{cases}$$

It is clear that if this definition is successful, M_n will be a full valuation of S. Furthermore, for each clause E in S, M_n will evaluate at least one literal of E to true (otherwise M_n would have contradicted E). Hence M_n will be a model of every clause in S. Therefore M_n will be a model of S (which is what we sought).

What we have left to verify is that M_i is well-defined, i.e. that at least one of the cases above must always arise. The proof of this is by contradiction.

Assume that neither of the cases are true for some j, and that j is the smallest number such that neither case arise. Then

(i) $M_{j-1} \cup \{A_j\}$ contradicts some clause C of S, and

(ii) $M_{j-1} \cup \{\neg A_j\}$ contradicts some other clause D of S, and

(iii) M_{j-1} does not contradict any clause of S (by the minimality of j).

From this we can conclude that C contains $\neg A_j$ and nothing but complements of M_{j-1}. Otherwise it would not have been contradicted by M_j. The same goes for D, except that it contains A_j instead of $\neg A_j$.
This also shows that $C \neq D$, because $A_j \notin C$, but $A_j \in D$.

Since $\neg A_j$ is in C, A_j is in D and $C \neq D$, we can form the resolvent

$$B = (C - \neg A_j) \cup (D - A_j)$$

B can contain nothing but complements to the literals of M_{j-1}, i.e. it is contradicted by M_{j-1}.
But B must be part of S since S is $\mathcal{R}$-satisfied. So M_{j-1} contradicts a clause B in S, which is impossible because of (iii) above. □

Example. Suppose we run the resolution procedure on the sentence

$$S = \langle [A \vee B] \wedge [\neg A \vee B] \wedge [\neg B \vee \neg C] \rangle$$

As a table, the successive addings of the $\mathcal{R}$ operation are:

S	$\mathcal{R}(S)$	$\mathcal{R}^2(S)$	$\mathcal{R}^3(S)$
$[A \vee B]$	$[B]$	$[\neg C]$	
$[\neg A \vee B]$	$[A \vee \neg C]$	$[B \vee \neg C]$	
$[\neg B \vee \neg C]$	$[\neg A \vee \neg C]$		

where we see that no new resolvents are found the third time we apply $\mathcal{R}$. Neither has the empty clause been added at any point, which means that $\mathcal{R}^2(S)$ is

$\mathcal{R}$-satisfied and that we should be able to find a model for $\mathcal{R}^2(S)$.

Start with $M_0 = \emptyset$, and the order A, B, C of the propositional letters. Since $\{A\}$ does not contradict any clause, we let $M_1 = \{A\}$. Moving on, we find that neither $\{A, B\}$ contradicts any clause, rendering to $M_2 = \{A, B\}$.

But $\{A, B, C\}$ contradicts for example $[\neg C]$ in the $\mathcal{R}^2(S)$ column, hence we are compelled to choose $M_3 = \{A, B, \neg C\}$.

Verifying M_3 against S shows that M_3 indeed satisfies every clause of S, and hence is a model of S.

To sum up we now know that our satisfiability checker will determine satisfiability of a given sentence in finite time. In fact, we even have an upper bound of the time consumption: 2^{2^l}. Also, if we wanted, we could use the algorithm in the proof of lemma 2.3 to return a satisfying valuation instead of just `satisfiable` in case the given sentence was satisfiable.

The fact that it always solves the problem is the most important aspect however, and as a grand finale we express this as a theorem:

Theorem 2.4 (Completeness)**.** *For any sentence S on normal form, the propositional resolution procedure will, in finite time, return the correct answer to the question whether S is satisfiable or not.*

3 Predicate Logic

In this section we will mainly be focusing on extending the resolution procedure to sentences of predicate logic.

The most striking difference between sentences of propositional and predicate logic is that the latter ones contain variables. In the normal form we will be using, all variables will be universally quantified. This enables us to search for resolvents not only from the formulas as they stand, but also from any instantiation of the universally quantified variables.

Consider, for instance, the clauses

$$[A(y)], [\neg A(f(a)) \vee B(a)]$$

$A(y)$ (in the first clause) and $\neg A(f(a))$ (in the second clause) are not complements. Therefore we can not apply resolution immediately. But the variable y is universally quantified, and can thus be *instantiated* with, or *substituted* for, any term. Substituting it for $f(a)$, for example, would yield a complementary pair; and thereby the resolvent $[B(a)]$.

A significant part of the theory developed will address the question of finding the right substitutions, i.e. substitutions yielding new and useful resolvents. Before that a number of definitions will be postulated; the relevant normal forms be discussed; and Herbrand's theorem, which is central to the theory of ATP for predicate logic, be proven.

3.1 Definitions

3.1.1 Syntax

Language The predicate language extends the propositional language with the following components:

- *Variables* $x, y, z, \ldots$, and *constants* $a, b, c \ldots$.
- *Functions* $f, g, h, \ldots$, or, more formally, with an integer k determining its *arity* (i.e. the number of arguments it takes). The function f^k takes k arguments. A function with arity 0 is effectively a constant.
- *Predicate letters* $A, B, C, \ldots$, also together with an integer determining their arity when such precision is advantageous. A^k is a predicate letter taking k arguments. A predicate letter of arity 0 is essentially a propositional letter.

All of which may be indexed with a natural number when a greater number of symbols are required.

Finally, the *quantifiers* $\forall, \exists$ are also added to the language.

Term. A *term* is either a variable or a constant, or a function of arity k with terms $t_1, \ldots, t_k$ as arguments. For example $f^k(t_1, \ldots, t_k)$, although we will often omit the k and just write $f(t_1, \ldots, t_k)$.

We will often denote terms with t or s.

Atomic Formula. An *atomic formula* is a predicate letter of arity k together with k terms. For instance $A^k(s_1, \ldots, s_k)$, or simply $A(s_1, \ldots, s_k)$.

Formula. A *formula* is either (i) an atomic formula, or (ii) one or two formulas combined with a propositional connective, or (iii) a quantifier followed by a variable followed by a formula.

For example $A(t)$, $(P \vee Q)$, or $\forall x P$ (where t is a term, A a unary predicate letter, and P and Q formulas).

Literal. A *literal* is either an atomic formula, or the negation of an atomic formula.

Example: $A(t)$, $\neg B(s_1, s_2, s_3)$, but not $(A(t) \wedge B(s_1, s_2, s_3))$.

Tightly connected to this is the concept of the *complement* of a literal L, written $\overline{L}$. For an atomic formula A, the complement of A is $\neg A$, and the complement of $\neg A$ is A.

L and $\overline{L}$ are *complements*, they form a *complementary pair*.

Closed and Grounded. An *expression* E (i.e. either a term or a formula) is considered *grounded* if it contains no variables. And E is *closed* if all variables in E are bound by some quantifier.

For instance, the term $f(g(x, y))$ is not grounded since it contains the variables x and y. The formula $\forall x A(f(x, a))$ is closed (since x is bound), but it is not grounded since it contains a variable x.

For terms there is no difference between closed and grounded. And for formulas, groundedness implies closedness.

Generalized Conjunction and Disjunction, Clause and Dual Clause. The definitions and notations of these are identical to the propositional case. See section 2.1.

3.1.2 Semantics

Model. A *model* M is an ordered pair $\langle D, I \rangle$ where D is a nonempty set called the *domain*, and I an *interpretation*, i.e. a function that maps:

- each constant c to an element c^I in D,
- each k-ary function symbol f^k to a function $(f^k)^I : D^k \to D$, and
- each k-ary predicate letter A^k to a k-ary relation on D, (which can be identified with its *extension*, i.e. a subset $(A^k)^I \subseteq D^k$).

We shall also let $f(t_1, \ldots, t_n)^I$ denote $f^I(t_1^I, \ldots, t_n^I)$, i.e. the object f^I takes on input $(t_1^I, \ldots, t_n^I)$, and let $A(t_1 \ldots, t_n)^I$ be true if and only if $(t_1^I, \ldots, t_n^I)$ satisfies the relation A^I (is an element in the extension A^I).

Assignment. When free variables are involved, we need to extend the interpretation with an assignment.

An *assignment* in a model $M = \langle D, I \rangle$ is a function A that maps each free variable x to an element $x^A \in D$.

Satisfiability. A formula P is *satisfiable* if it is satisfied by some model. For example is $\forall x A(x)$ satisfiable, since it is satisfied by the model $M = \langle \{1, 2, \dots\}, I \rangle$, where I assigns the universal relation to A (i.e. $A^I = D$).

A formula is *valid* if it is satisfied by every model.

Two formulas are *equivalent* if they are satisfied by exactly the same models. Two formulas P and Q are *equisatisfiable* if: P satisfiable $\iff Q$ satisfiable.

3.2 Normal Forms

Ultimately we want to work with formulas on the form

$$\forall x_1 \dots \forall x_n \langle [L_{11} \vee \dots \vee L_{1k_1}] \wedge \dots \wedge [L_{m1} \vee \dots \vee L_{mk_m}] \rangle$$

where each L_{ij} is a literal that contains no other variables than $x_1, \dots, x_n$.

Of great importance is that the clauses contains no quantifiers, and that only universal quantifiers remain. The latter means that we cannot find an *equivalent* formula on normal form for all formulas, only an *equisatisfiable* one. That will suffice though, since our method only determines satisfiability anyway. (To have it determine validity, negate the input. See page 6.)

The conversion of a formula P_1 to normal form, can be divided into steps. First we convert it to an equivalent formula P_2 on *negation normal form*. Then we move all quantifiers in P_2 to the front, making it *prenex*.

The result P_3 is then *skolemized*, in order to get rid of the existential quantifiers. Finally, the result of that, P_4, is converted to a clause form formula P_5.

P_5 will then be on the desired form, as we shall see.

3.2.1 Negation Normal Form

A first order formula is on *negation normal form* when all negation signs prefixes atomic formulas. To achieve that we can simply extend the algorithm we used in the propositional case (cf. section 2.2.1), with rules for the quantifiers.

The rules, building on tautologies of predicate logic, are

- $\neg \forall x P$ converts to $\exists x \neg P$, and
- $\neg \exists x P$ converts to $\forall x \neg P$.

Apart from these additions, the method is identical to the propositional one.

Example. The formula $\neg \forall x \exists y (A(x) \vee B(y))$ becomes:

1. $\neg \forall x \exists y (A(x) \vee B(y))$
2. $\exists x \neg \exists y (A(x) \vee B(y))$
3. $\exists x \forall y \neg (A(x) \vee B(y))$
4. $\exists x \forall y (\neg A(x) \wedge \neg B(y))$

3.2.2 Prenex Form

The next step of the conversion is to move all quantifiers to the front. This is rather easy to do on a formula already on negation normal form, since we can simply "move out" quantifiers due to the equivalences

- $(\forall x P \wedge Q) \Leftrightarrow \forall x(P \wedge Q)$,
- $(Q \wedge \forall x P) \Leftrightarrow \forall x(Q \wedge P)$,
- $(\forall x P \vee Q) \Leftrightarrow \forall x(P \vee Q)$,
- $(Q \vee \forall x P) \Leftrightarrow \forall x(Q \vee P)$,

which are all the types of subformulas we may encounter in a negation normal form formula.

The equivalences are however only valid if Q does not contain x. In the event that Q would contain x, we have to do a a *variable substitution* first, as explained in the following example.

Example. Suppose we want to convert $\exists x_1(A(x_1) \wedge \forall x_1 B(x_1))$ to prenex form. Directly moving out the $\forall$ quantifier, would yield the *non*-equivalent formula $\exists x_1 \forall x_1(A(x_1) \wedge B(x_1))$.

If we instead change the variable x_1 to another variable x_2 in the subformula $\forall x_1 B(x_1)$, we would get (the equivalent formula)

$$\exists x_1(A(x_1) \wedge \forall x_2 B(x_2))$$

where we can safely move out the $\forall$ quantifier, yielding

$$\exists x_1 \forall x_2(A(x_1) \wedge B(x_2))$$

which is equivalent to the original formula $\exists x_1(A(x_1) \wedge \forall x_1 B(x_1))$ and on prenex form.

The idea is that whenever we have a quantifier binding x, we have to check that x is not bound by another quantifier within the scope of the first one. If x is bound by an inner quantifier, we change the variable of the inner quantifier (together with all occurrences bound by the inner quantifier) to a new variable y.

Substituting a variable this way has no semantical impact at all, so the formula with a substituted variable is always equivalent to the original one.

3.2.3 Skolem Form

The goal is to get rid of all existential quantifiers. By successively exchanging existential quantifiers to new function symbols we get an equisatisfiable formula on *skolem form* (it will not always be equivalent, see example below).

A formula without existential quantifiers is on *skolem form*. The process of successively exchanging quantifiers for function symbols is called *skolemization*.

Lemma 3.1. *The formula* $\forall x_1 \ldots \forall x_n \exists y P(x_1, \ldots, x_n, y)$ *is equisatisfiable with* $\forall x_1 \ldots \forall x_n P(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$ *(as long as* P *does not contain the function symbol* f*).*

Proof. We will show that if we have a model satisfying $\forall x_1 \dots \forall x_n \exists y P(x_1, \dots, x_n, y)$, we can transform that into a model satisfying $\forall x_1 \dots \forall x_n P(x_1, \dots, x_n, f(x_1, \dots, x_n))$. And vice versa.

Assume that $\forall x_1 \dots \forall x_n \exists y P(x_1, \dots, x_n, y)$ is satisfied by some model $M_1 = \langle D, I_1 \rangle$. Then $\forall x_1 \dots \forall x_n P(x_1, \dots, x_n, f(x_1, \dots, x_n))$ is satisfied by some model $M_2 = \langle D, I_2 \rangle$, where I_2 only differ from I_1 in the interpretation of f.

By assumption, we can for any n-tuple $d_1, \dots, d_n \in D^n$ find an $e \in D$ such that $P^{I_1}(d_1, \dots, d_n, e)$ is satisfied. This means that we can express the choice of e as a function of $d_1, \dots, d_n$: for any choice of $d_1, \dots, d_n$, let $e_{d_1,\dots,d_n}$ be an element $e \in D$ such that $P^{I_1}(d_1, \dots, d_n, e_{d_1,\dots,d_n})$ is satisfied.

Now, let I_2 be as I_1 except for the interpretation of f. In fact, let $f^{I_2}(d_1, \dots, d_n) = e_{d_1,\dots,d_n}$ for every $d_1, \dots, d_n \in D^n$.

Then $P(x_1, \dots, x_n, f(x_1, \dots, x_n))$ must be satisfied by M_2.

The converse is more straightforward. Assume

$$\forall x_1 \dots \forall x_n P(x_1, \dots, x_n, f(x_1, \dots, x_n))$$

is satisfied by $M_2 = \langle D, I_2 \rangle$. Then $P^{I_2}(d_1, \dots, d_n, f^{I_2}(d_1, \dots, d_n))$ for all choices of $d_1, \dots, d_n$.

Hence M_2 also satisfies $\forall x_1 \dots \forall x_n \exists y P(x_1, \dots, x_n, y)$. Choose namely an n-tuple $d_1, \dots, d_n \in D^n$. Then we can find an e such that $P^{I_2}(d_1, \dots, d_n, e)$, namely $e = f^{I_2}(d_1, \dots, d_n)$. But the choice of $d_1, \dots, d_n$ was arbitrarily made, hence we can find an e for every choice of $d_1, \dots, d_n$. □

Having proved the lemma, it is now easy to see that any formula P can be converted into an equisatisfiable formula on skolem form.

By first converting P to negation normal form and prenex form, it is easy to see that successively applying the lemma until all existential quantifiers are gone, will render a formula on skolem form.

Example. Skolemizing the the formula $\forall x(\neg \forall y A(x, y) \vee \exists z A(x, z))$. The first step is to rewrite it on negation normal form and prenex form:

$$\forall x \exists y \exists z(\neg A(x, y) \vee A(x, z))$$

This make us go clear of the potential difficulty of the second universal quantifier essentially being an existential quantifier (due to the preceding negation sign).

The skolemization is then straightforward:

1. $\forall x \exists y \exists z(\neg A(x, y) \vee A(x, z))$
2. $\forall x \exists z(\neg A(x, f_1(x)) \vee A(x, z))$
3. $\forall x(\neg A(x, f_1(x)) \vee A(x, f_2(x)))$

It is easy to see that the last formula is *not* equivalent to the first one. The formulas are only equisatisfiable.

3.2.4 Clause Form

The *clause form* (or *clause normal form*) for predicate logic is just like the clause form for propositional logic, with the addition that quantifiers should be outside the generalized conjunction.

The argument that any prenex formula can be converted to clause form is essentially identical to the argument for propositional formulas. Neither the conversion procedures require any substantial modifications (cf. 2.2.3).

Example. $\forall x \exists y \langle [A(x,y) \vee B(x)] \wedge [A(x,x)] \rangle$ is a formula on clause normal form.

3.2.5 Normal Form

Since sentences on negation-, prenex-, clause- and skolem normal form will be so frequently used, we will simply say that any such formula is on *normal form*. It should be clear from the discussion above that we can rewrite any formula P on normal form with maintained satisfiability. We state that in a theorem for future reference.

Theorem 3.2. *For any formula P there is an equisatisfiable formula P' on normal form.*

Normally we will only consider formulas with all variables bound (i.e. closed formulas), and since the only type of quantifier in a normal form formula is the universal one, we will sometimes adopt the convention of dropping the quantifiers. This is no loss of information, we know that all variables are universally quantified.

We will also make frequent use of the equivalence

$$\forall x_1 \ldots \forall x_m \langle P_1 \wedge \cdots \wedge P_n \rangle \Leftrightarrow \langle (\forall x_1 \ldots \forall x_m P_1) \wedge \cdots \wedge (\forall x_1 \ldots \forall x_m P_n) \rangle$$

The right form is especially useful when we wish to instantiate one of the clauses of a formula separately, which is a rather common situation when dealing with resolution.

Example. The normal form formula $\langle [A \vee B(x,y)] \wedge [B(z,z)] \rangle$ is understood to mean

$$\forall x \forall y \forall z \langle [A \vee B(x,y)] \wedge [B(z,z)] \rangle$$

Or, if more useful,

$$\langle (\forall x \forall y \forall z [A \vee B(x,y)]) \wedge (\forall x \forall y \forall z [B(z,z)]) \rangle$$

3.3 Herbrand's Theorem

Normally we make a clear distinction between the *syntactical* and the *semantical* aspects of our logic. The syntactics is about the *language* as a *system of symbols*. Alphabets and grammars, the criteria for a well formed formula, free variables and substitutions, are all syntactical features.

To the semantics belong concepts such as interpretations, instantiations and valuations, domains, validity and satisfiability: essentially anything that is related to the *meaning* of the formulas.

We may for example have a formal theory about algebraic groups, about the natural numbers, or about any other (non-logical) structure. When we then make deductions in the theory we say that we derive properties about the structure that it is about. If we have a proof in a formalization of the natural numbers, we take a proof in that theory to be a proof of a certain property of natural numbers.

Herbrand's theorem essentially states that whenever we have a formula (or a set of formulas) we never need to interpret it to talk about something non-logical. Rather, we can always take it to talk about its own terms. We can interpret it under a *Herbrand model*.

3.3.1 Herbrand Models

Formally, we define the *Herbrand universe* H_S of a sentence S, as the set of grounded terms that we can create from the *function set* of S. The function set F_S is the set of all constants, parameters and functions in S, if there is at least one constant or parameter in S. Otherwise it is the set of functions of S together with a new constant a.

Example. The Herbrand universe of $\forall x A(f(x, c))$ is

$$\{c, f(c), f(f(c)), \dots\}$$

(the function set is $\{c, f\}$).

Example. The function set of

$$B(f(x), g(x, y))$$

is $\{a, f, g\}$. Hence we get the Herbrand universe

$$\{a, f(a), g(a, a), f(f(a)), f(g(a, a)), g(a, f(a)), g(a, g(a, a)) \dots\}$$

Here we see the reason of adding the parameter a to the function set. Had we not done so, the Herbrand universe would have been empty.

Definition. A finite subset of a Herbrand universe is called a *Herbrand domain*.

The concept of a Herbrand universe extends easily to a set of sentences or a whole language. Simply let S stand for a set or a language instead of a sentence in the above definition.

The Herbrand universe of a language is the base of the definition of Herbrand model.

Definition. A *Herbrand model* for a language L is a model $\langle H, I\rangle$ such that

1. the domain H is the Herbrand universe of L, and
2. the interpretation I maps every grounded term t to itself (i.e. $t^I = t$).

Such models have several advantages:

- Interpretations become rather redundant, every sentence is already its own interpretation. The only thing an interpretation does is specifying relations (extensions) for the predicate letters.
- Assignments are simply a special form of substitutions (substitutions to grounded terms).
- The interpretation of the quantifiers becomes very straight forward:
  - $\forall x A(x)$ is true if and only if $A(t)$ is true for every grounded term t,
  - $\exists x A(x)$ is true if and only if $A(t)$ is true for some grounded term t.

So already here we see how the concepts of Herbrand universes and models tie semantical concepts to syntactical. Herbrand models make the language talk not of objects of some other sphere, but of the terms of the language. Universal and existential statements become statements about the existence of certain grounded terms.

This will pave the the road for *Herbrand's theorem* which reduces satisfiability problems of predicate logic to satisfiability problems of propositional logic. The latter having the nice feature of being decidable.

We will now have a closer look at some of the properties of Herbrand models.

3.3.2 Properties of Herbrand Models

The first important property of Herbrand models, is that whenever there is any model satisfying a formula, there is also a Herbrand model doing the same job. That means that if we are interested in the determining the satisfiability of a formula, it will be enough to investigate whether there is any Herbrand model satisfying it. If we show that no such Herbrand model can exist, we have shown that no satisfying model at all exists, so the sentence must be contradictory.

Theorem 3.3. *A formula P is satisfiable if and only if there is a Herbrand model satisfying it.*

Proof. By theorem 3.2 we can assume that P is on normal form, and that it is satisfied by some model $M_D = \langle D, I_D \rangle$. We shall show that there is a Herbrand model $M_H = \langle H, I_H \rangle$ satisfying P (where H is the Herbrand universe of P).

Define M_H the following way:

- for each grounded term t, define $t^{I_H} = t$,
- for each predicate letter A^k, define the extension of

$$(A^k)^{I_H} = \{(t_1, \ldots, t_k) \in H : A^k(t_1, \ldots, t_k)^{I_D}\}$$

  (remember that $A^k(t_1, \ldots, t_k)^{I_D}$ means that $(t_1^{I_D}, \ldots, t_k^{I_D})$ satisfies the relation $(A^k)^{I_D}$, cf. section 3.1).

From the definition of M_H it follows that for any grounded atomic formula Q, then Q^{I_H} if and only if Q^{I_D}.

For assume that $A(t_1, \ldots, t_k)$ is a grounded atomic formula that is true under M_D. Then $(t_1^{I_D}, \ldots, t_k^{I_D})$ is in the extension of A^{I_D}, and hence $(t_1^{I_H}, \ldots, t_k^{I_H}) = (t_1, \ldots, t_k)$ is in the extension of A^{I_H}, according to the definition.

Conversely assume that $A(t_1, \ldots, t_k)$ is not true under M_D. Then $(t_1, \ldots, t_k)$ will not be in the extension of A^{I_H}.

From this it is immediate that all *propositional combinations* Q of grounded atomic formulas (i.e. any grounded formula Q not containing quantifiers), will have the same truth value in M_D and M_H.

Now we only have one case left to consider, the universally quantified formula $\forall x_1 \ldots \forall x_n Q(x_1, \ldots, x_n)$, where $Q(x_1, \ldots, x_n)$ is quantifier-free and contains no other free variables than $x_1, \ldots, x_n$. Assume that $\forall x_1 \ldots \forall x_n Q(x_1, \ldots, x_n)$ is satisfied by M_D. Then the implication

$$\forall x_1 \ldots \forall x_n Q(x_1, \ldots, x_n) \implies Q(t_1, \ldots, t_n)$$

gives that $Q(t_1, \ldots, t_n)^{I_D}$ for all grounded terms $t_1, \ldots, t_n$.

Since Q contains no quantifiers, we know that

$$Q(t_1, \ldots, t_n)^{I_D} \iff Q(t_1, \ldots, t_n)^{I_H}$$

from above.

Hence $Q(t_1, \ldots, t_n)^{I_H}$ for all grounded terms $t_1, \ldots, t_n$, and thereby also $(\forall x_1 \ldots \forall x_n Q(x_1, \ldots, x_n))^{I_H}$. Which was to be proven. □

Herbrand Expansion. For a sentence $P(x_1, \ldots, x_n)$ that contains no other free variables than $x_1, \ldots, x_n$, the *Herbrand expansion* over a set D of grounded terms is the set $\{P(t_1, \ldots, t_n) : t_i \in D\}$, denoted $\mathcal{E}(P, D)$.

Example. $A(x)$ expanded over $D = \{a, f(a)\}$ is $\mathcal{E}(A, D) = \{A(a), A(f(a))\}$.

In the case of a normal form formula $P(x_1, \ldots, x_n)$ (where we normally take the variables to be implicitly universally quantified), we still let the Herbrand expansion $\mathcal{E}(P(x_1, \ldots, x_n), D) = \{P(t_1, \ldots, t_n) : t_i \in D\}$.

By means of a Herbrand expansion we can express the fact that a sentence $\forall x_1, \ldots, x_n Q(x_1, \ldots, x_n)$ is true in a Herbrand model M if and only if $Q(t_1, \ldots, t_n)$ is true in M for all $t_i \in H_Q$.

(This is an immediate consequence of the definition of the universal quantifier. A universally quantified formula is true if and only if the formula is true for every element in the domain.)

Proposition 3.4. *Assume $Q(x_1, \ldots, x_n)$ is a sentence containing no other variables then $x_1, \ldots, x_n$. Then $\forall x_1 \ldots \forall x_n Q(x_1, \ldots, x_n)$ is true in a Herbrand model M if and only if M satisfies (every member of) $\mathcal{E}(Q, H_Q)$.*

3.3.3 Herbrand's Theorem

We are now ready for the main result of this section:[2]

[2] A more common version of Herbrand's theorem is the equivalent: $\exists x_1, \ldots, x_n A(x_1, \ldots, x_n)$ is valid if and only if $[A(t_{11}, \ldots, t_{n1}) \vee \cdots \vee A(t_{1k}, \ldots, t_{nk})]$ is valid for some $t_{ij} \in H_S$ and $k \in \mathbb{N}$.

Theorem 3.5 (Herbrand's Theorem)**.** *Consider a formula P on normal form (with dropped quantifiers). Then P is unsatisfiable if and only if some finite subset $K \subseteq \mathcal{E}(P, H_P)$ is unsatisfiable.*

Proof. By theorem 3.3 we have that P is satisfiable if and only if P is satisfiable by some Herbrand model.

That means that we from proposition 3.4 can draw the conclusion: P is satisfiable if and only if $\mathcal{E}(P, H_P)$ is satisfiable.

And the compactness theorem gives that a set $\mathcal{E}(P, H_P)$ is satisfiable if and only if every finite subset of $\mathcal{E}(P, H_P)$ is satisfiable. (See [3, p. 132] for a proof of the compactness theorem.)

Combined and rephrased: P is unsatisfiable if and only if some finite subset $K \subseteq \mathcal{E}(P, H_P)$ is unsatisfiable. □

Corollary 3.6. *A normal form formula with dropped quantifiers P is unsatisfiable if and only if $\mathcal{E}(P, D_P)$ is unsatisfiable for some Herbrand domain D_P of P.*

Proof. It is immediate that every $\mathcal{E}(P, D_P)$ is satisfiable if P is.

Conversely assume that P is unsatisfiable. Then theorem 3.5 gives a finite unsatisfiable $K \subseteq \mathcal{E}(P, H_P)$. But K is included in $\mathcal{E}(P, D_P)$ for $D_P =$ "the set of grounded terms occurring in K" (which is a Herbrand domain of P).

And since $\mathcal{E}(P, D_P)$ contains an unsatisfiable subset K, then $\mathcal{E}(P, D_P)$ must itself be unsatisfiable, which was to be proven. □

3.3.4 The Davis-Putnam Procedure

The Davis-Putnam procedure[2] is the proof procedure almost immediately suggested by corollary 3.6. (The version of the Davis-Putnam procedure that is described here does not follow the original version very closely. The principle is the same however.)

The corollary states that we can reduce the satisfiability problem of any sentence, to finite sets of grounded sentences. And a grounded set of sentences is essentially a set of sentences of propositional logic.

We can namely read each atomic formula $A(t_1, \ldots, t_k)$ (where $t_1, \ldots, t_k$ are closed terms) as a propositional letter. A Herbrand model M will then turn into a valuation of these "propositional letters": $A(t_1, \ldots, t_k)$ will be evaluated to true if $(t_1, \ldots, t_k)$ is in the extension of A^M, and false otherwise.

From this it follows easily that a set of grounded sentences is essentially a set of propositional sentences, just apply the same argument to every sentence in the set.

Hence it is as easy to determine satisfiability for a set of grounded sentences, as it is to determine satisfiability for a propositional sentences. We state this as a proposition, as it will come to use later as well.

Proposition 3.7. *The satisfiability problem of (a set of) grounded sentence(s), is essentially the same as the satisfiability problem for the propositional counterpart.*

So to show that a formula P is unsatisfiable, we only need to systematically go through all Herbrand domains D_P (which are countably infinitely many), and check whether $\mathcal{E}(P, D_P)$ is satisfiable or not.

Once we find a D_P such that $\mathcal{E}(P, D_P)$ is unsatisfiable, we have proven P to be unsatisfiable. Such a D_P is called a *proof set*. Corollary 3.6 gives that there is such a D_P if P is unsatisfiable, so as long as we systematically go through the Herbrand domains, we will find a proof.

A possible, reasonably efficient, way to go through all Herbrand domains of P is this (where F_P is the function set of P):

- Let $D_0 =$ "the constants of F_P".
- Let $D_{i+1} = D_i \cup$ "the terms we can create applying the functions of F_P to terms of D_i".

This will generate an (infinite) sequence of Herbrand domains. It is also clear that each finite subset of H_P will be included in some D_i, which is sufficient to eventually find a proof set if there is one (all supersets of a proof set are also proof sets).

As all proof procedures of predicate logic, this procedure is only guaranteed to stop if it is given an unsatisfiable formula (due to Turing's theorem 1.1 a proof procedure cannot be guaranteed to stop for all sentences). The major drawback of the Davis-Putnam procedure, is that the D_i:s might get quite big before we find a proof set. Especially when the function set contains functions of higher arity.

The resolution procedure (discovered shortly after the Davis-Putnam procedure) features a more sophisticated search procedure, making it more efficient in many situations. The main difference between the two is the *unification algorithm*, described below.

3.4 Substitutions

A *substitution* is a set $\{t_1/x_1, \ldots, t_k/x_k\}$. The *variable components* $x_1, \ldots, x_n$ are pairwise distinct, and $t_1, \ldots, t_k$ are terms (called the *term components*). The t_i/x_i:s are called *substitution components*, and must fulfill $x_i \neq t_i$.

Applying a substitution $\sigma = \{t_1/x_1, \ldots, t_k/x_k\}$ to a sentence S (or term u) is to replace each occurrence of the variable x_i with the term t_i. (In this essay we will only be interested in applying substitutions to normal form formulas with dropped quantifiers, so we do not need to worry about bound variables.) The result is written $S\sigma$ (or $u\sigma$).

Two substitutions are considered equal if they have the same effect on all expressions.

The possibility of some term t_i containing a variable x_j, requires us to be quite clear on the point that we *do not* first apply the substitution $\{t_1/x_1\}$, and then, *on the result*, apply $\{t_2/x_2\}$ etc. Rather we apply them all "simultaneously". For example

$$\langle [A(x, y)] \rangle \{y/x, b/y\} = \langle [A(y, b)] \rangle$$

which is not to be confused with

$$(\langle [A(x, y)] \rangle \{y/x\})\, \{b/y\} = \langle [A(y, y)] \rangle \{b/y\} = \langle [A(b, b)] \rangle$$

This makes it, of course, extra important that no variable occurs twice in a substitution.

3.4.1 Combination

The combination of two substitutions σ, λ can be identified with one set. After all, the combination only have one impact on each variable.

Assume $\sigma = \{t_1/x_1, \ldots, t_m/x_n\}$ and $\lambda = \{s_1/y_1, \ldots, s_m/y_m\}$. Then, quite naturally,

$$\sigma\lambda = \{t_1\lambda/x_1, \ldots, t_n\lambda/x_n\} \cup \lambda'$$

where λ' is a subset of λ containing only those s_i/y_i such that $y_i \neq x_j$ for every j between 1 and n. Also, any substitution components u_i/z_i where $u_i = z_i$, is removed from $\sigma\lambda$.

Example. If, as in the above example, we have $\sigma = \{y/x\}$ and $\lambda = \{b/y\}$, we would get $\sigma\lambda = \{y\{b/y\}/x\} \cup \{b/y\} = \{b/x, b/y\}$.

This combination is transitive both in the sense that $(\sigma\lambda)\theta = \sigma(\lambda\theta)$, and that $(P\sigma)\lambda = P(\sigma\lambda)$ and $(t\sigma)\lambda = t(\sigma\lambda)$ for any formula P or term t.

Starting with the latter, it is enough to verify that for any variable x and substitutions σ, λ, then $(x\sigma)\lambda = x(\sigma\lambda)$. Depending on whether x is substituted by σ or not, the following cases arise.

1. If x is substituted for some term $t = x\sigma$ by σ, then $(x\sigma)\lambda$ will become $t\lambda$. This is exactly how $x(\sigma\lambda)$ is defined.
2. If x is not substituted by σ, then $(x\sigma)\lambda$ will simply be $x\lambda$. This also agrees with $x(\sigma\lambda)$. If x is substituted by λ but not by σ, x will be substituted by a substitution component from λ' in the substitution $\sigma\lambda$.

This proves the following proposition.

Proposition 3.8. *For P formula, t term, and σ, λ substitutions: $(P\sigma)\lambda = P(\sigma\lambda)$ and $(t\sigma)\lambda = t(\sigma\lambda)$.*

Corollary 3.9. *For σ, λ, θ substitutions, $(\sigma\lambda)\theta = \sigma(\lambda\theta)$.*

Proof. Take an arbitrary formula P. Then, from proposition 3.8, we have: $P((\sigma\lambda)\theta) = (P(\sigma\lambda))\theta = ((P\sigma)\lambda)\theta = (P\sigma)(\lambda\theta) = P(\sigma(\lambda\theta))$. Hence $(\sigma\lambda)\theta$ have the same effect as $\sigma(\lambda\theta)$ on every variable, and hence they must be equal. □

3.5 Unification

The ingenious trick of the resolution procedure is to instead of fully instantiating the given formula (with ground terms) over and over again, only "pseudo-instantiate" the formula with terms still containing free variables, but are specific enough to render a resolvent.

A resolvent found in the Davis-Putnam procedure is final since it only contains grounded terms, but a resolvent of the resolution procedure can be used for further instantiation.

Example. Consider the (somewhat artificially chosen) sentence

$$S = \langle C_1 \wedge C_2 \wedge C_3 \rangle = \langle [A(x) \vee \neg B(x)] \wedge [\neg A(f(x))] \wedge [B(f(g(x, y)))] \rangle$$

First applying the substitution $\sigma = \{f(x)/x\}$ to the first clause, makes C_1 become $C_1\sigma = [A(f(x)) \vee \neg B(f(x))]$.

From $C_1\sigma = [A(f(x)) \vee \neg B(f(x))]$ and $C_2 = [\neg A(f(x))]$, we find the resolvent $R_1 = [\neg B(f(x))]$.

Applying a new substitution $\tau = \{g(x,y)/x\}$ to R_1, gives us the empty clause as resolvent from $R_1\tau = [\neg B(f(g(x,y)))]$ and $C_3 = [B(f(g(x,y)))]$.

This should give an idea of the gained efficiency of only "pseudo-instantiating" free variables. Had we in the example chosen σ to be a ground term, and applied a similar substitution to C_2 to get our resolvent, we would not have been able to continue from R_1 to get the empty clause. Instead we would have had start over with a more complicated substitution to find the empty clause.

The idea to instantiate "as little as possible" in order to have the utmost freedom left to find another resolvent, is perhaps the main benefit of the resolution procedure. The unification algorithm is used to find the least limiting substitution available (called *the most general unifier* below), that still enables application of the resolution rule.

The principal example of its usage is when we have a formula $A(t)$ in one clause, and, in another, $\neg A(s)$. We then ask ourselves, is there a substitution σ that *unifies* $A(t)$ with $A(s)$, i.e. a σ such that $A(t)\sigma = A(s)\sigma$? Such a substitution is what is required to infer a resolvent.

In the general case however, it is not just two formulas we want to unify, but several. Hence we will develop the unification algorithm for a finite set N of formulas.

Definition. We say that a set N of formulas is *unified* by a substitution σ if $N\sigma$ is a singleton, i.e. for every $P_1, P_2 \in N$, $P_1\sigma = P_2\sigma$. Any substitution λ that unifies N is a *unifier* for N.

Example. Let $N = \{A(x), A(f(y)), A(f(f(z)))\}$. Then N is unified by the substitution $\sigma_N = \{f(f(z))/x, f(z)/y\}$ since for every P in N so is $P\sigma_N = A(f(f(z)))$. Hence $N\sigma_N = \{A(f(f(z)))\}$.

A special kind of unifier is the *most general unifier*. A most general unifier for a set N is a unifier λ_N for N such that if θ also unifies N, then $\theta = \lambda_N\sigma$ for some substitution σ.

3.5.1 Disagreement set.

For two formulas to be identical, they need to comprise the same predicate letter and the same arguments. The same goes for terms: they need to share function letter, and share term arguments.

Say that we want to unify $A(f(x))$ with $A(f(f(y)))$. Then the formulas do agree on predicate letters, but have (slightly) disagreeing arguments. The arguments in turn (i.e. $f(x)$ and $f(f(y))$) agree on function letter, but have disagreeing arguments.

Moving down another level, we find that the arguments of the arguments (i.e. x and $f(y)$), do disagree on their function letter (if we consider x its own function letter). Intuitively this is our *disagreement set*. That is, a "highest" set such that the function letters (or predicate letters) disagree, is a disagreement set.

Unifying all disagreement sets of a set of formulas Γ, will unify Γ.

Definition. We call, for an atomic formula $A(t_1, \ldots, t_n)$ or term $f(t_1, \ldots, t_n)$, the t_i:s the *arguments* (of the expressions) and A and f the *expression letters*. Constants, parameters and variables have zero arguments, and are their own expression letters. Predicate and function letters, constants and parameters are called *unmodifiable letters*, since they are unaffected by substitutions.

Expressions may differ in respect of their expression letter or in respect of their arguments.

Example. $A(f(x))$ is different from $A(g(x))$ with respect to their argument, whereas $f(x)$ and $g(x)$ are different with respect to their expression letter (but have identical arguments).

In our unification algorithm we will be interested finding a *disagreement set* for a set of expressions S containing at least two elements. This is the job of the recursively defined procedure Algorithm 2.

Algorithm 2 Disagreement-Set(S)

```
if the expression letters of the members of S are not all identical then
  return S
else
  pick a k such that the k:th argument is not the same for all members of S
  return the result of Disagreement-Set(the set of the k:th arguments).
end if
```

A *disagreement set* for S is simply any set that can be returned by the above procedure applied to S.

Example. Find a disagreement set for the set $S = \{A(x, y), A(x, x), A(f(z), g(x, f(z)))\}$. The expressional letter is A for each member of S. Hence we shall look for an argument for which they disagree. Since they disagree on the first argument, we apply the procedure once again on the set $\{x, x, f(z)\} = \{x, f(z)\}$. This time the members do not share expressional letter, and so we have found our disagreement set: $\{x, f(z)\}$.

3.5.2 Unification Algorithm

With the concept of disagreement set available the unification algorithm is ready to be defined (Algorithm 3). The idea is a rather simple one: Starting with a finite nonempty set N and an empty substitution σ we successively update σ to unify more and more disagreement sets (the variable U in the algorithm).

Each pass through the while loop will update σ to unify two terms of a disagreement set. Eventually, either all disagreement sets will be unified and σ will be the unifier we seek; or a disagreement set which is not unifiable will be found, in which case we return `fail`. The correctness of the algorithm will be proven below.

Example. Consider $N = \{A(x), A(f(y)), A(f(f(z)))\}$ as in an example above. Applying the unification algorithm on N, we first check whether $N\{\}$ is a singleton (line 2). As it is not a singleton we enter the while loop.

Algorithm 3 Unification-Algorithm(N)

```
1: σ ← ∅
2: while Nσ is not a singular do
3:   U ←Disagreement-Set(Nσ)
4:   if U contains no variable then
5:     return fail
6:   else
7:     pick a variable v from U,
8:     pick a term t from U − v,
9:     if x occurs in t then
10:      return fail,
11:    else
12:      σ ← σ{t/v}
13:    end if
14:  end if
15: end while
16: return σ.
```

Finding the disagreement set $U = \{x, f(y), f(f(z))\}$, we proceed to line 4. Passing the fail criteria, we choose the variable x (line 7), and pick the term, say, $f(y)$ (line 8). Accordingly, σ updates to $\sigma = \{f(y)/x\}$ (line 9).

Starting the while-loop over with the new σ, we get: The disagreement set for $N\sigma = \{A(f(y), A(f(f(z)))\}$ becomes $U = \{y, f(z)\}$. Hence y will become our variable, and $f(z)$ our term. σ updates to $\sigma\{f(z)/y\} = \{f(y)/x\}\{f(z)/y\} = \{f(f(z))/x, f(z)/y\}$ according to the combination rule.

We now see that $N\sigma = N\{f(f(z))/x, f(z)/y\} = \{A(f(f(z)))\}$ is a singleton, which means that we are done. N was unifiable. $\sigma = \{f(f(z))/x, f(z)/y\}$ is returned (line 16).

Example. Let us also consider an ununifiable example: $N = \{A(x), A(f(x))\}$. At line 3 the disagreement set would be $U = \{x, f(x)\}$, rendering x the variable and $f(x)$ the term. This will make the algorithm return fail due to the fail criteria at line 9, the variable x occurs in $f(x)$.

It is clear that no substitution could ever have unified N, since whatever we would substitute x for, would then also occur inside $f(x)$.

To verify that the unification algorithm is correct, we wish to prove the following theorem:

Theorem 3.10. *For any finite nonempty set N of atomic formulas the unification algorithm returns a most general unifier σ_N if N is unifiable, and returns `fail` otherwise.*

Proof. The theorem follows directly from lemma 3.14 and 3.15 below. □

Lemma 3.11. *The algorithm will terminate in finite time.*

Proof. Assume the set $N\sigma$ have n distinct variables at one pass through the while loop. Then the next pass $N\sigma$ will have at most $n - 1$ distinct variables, for in each pass the variable v (chosen at line 7) will disappear from $N\sigma$.

And since we are starting with a finite number of variables in N (N must be a finite set), the algorithm could never go through the while loop an infinite number of times. □

Lemma 3.12. *Assume that N is a finite nonempty set of literals that is unified by λ, and assume that there is a disagreement set U for N. Let v be a variable in U and t a term in $U - v$. Then $\{t/v\}\sigma = \sigma$.*

Proof. σ unifies N by assumption, and must hence unify every disagreement set of N. So it unifies U, and thereby also v and t (i.e. $v\sigma = t\sigma$).

Now, $\{t/v\}\sigma = \sigma$ if and only if they have the same effect on all variables. It is clear that for any variable $x \neq v$, $x\{t/v\}\sigma = x\sigma$ (since $x\{t/v\} = x$). But we also have $v\{t/v\}\sigma = t\sigma = v\sigma$ (the last equality by assumption).

Hence we have proved the desired equality $\{t/v\}\sigma = \sigma$. □

Lemma 3.13. *If a disagreement set U for a literal set N contains no variables, or if one of the variables in U appears inside one of the non-variables in U, then U is not unifiable.*

Proof. Assume that U is a disagreement set that contains no free variables. Then the members of U do not all have the same expressional letter. And since substitutions only affect the expression letter of variables, no substitution can ever unify U.

Similarly, if a variable v of U is a proper subexpression of some term t in $U - v$, no substitution can ever unify v and t. No matter what substitution λ we choose, $v\lambda$ will always be a proper subexpression of $t\lambda$, hence $v\lambda \neq t\lambda$ for all λ, and so λ can not unify U. □

Lemma 3.14 (Main Lemma)**.** *The algorithm will terminate with a most general unifier, if presented a unifiable set N.*

Proof. Here we will introduce a *loop invariant*, i.e. a condition that is true each time we are about to enter the while loop. Assume that our input set N is unified by θ, and that σ is the substitution we want to make a most general unifier by successively extending it. Then our invariant is: $\theta = \sigma\theta$. The proof that our invariant is indeed an invariant (is true for each pass through the while loop) is by induction:

The invariant is obviously satisfied in the first pass, for then $\sigma = \emptyset$.

Assume that the invariant is satisfied in the k:th iteration and that $N\sigma$ is not a singleton. Then we can find a disagreement set U for $N\sigma$. Now, U must be unifiable if $N\sigma$ is, and $N\sigma$ is unified by θ since $(N\sigma)\theta = N(\sigma\theta) = N\theta$. Hence U is unifiable.

By Lemma 3.13 we then know that we can pick a variable v from U, and a term t from $U - v$, such that v does not occur in t. Thus v and t can be unified with $\{t/v\}$. Extending σ with $\{t/v\}$ gives $\sigma_{new} = \sigma\{t/v\}$.

Then it only needs to be verified that the loop invariant is still satisfied for the new substitution σ_{new}. But the loop invariant must be satisfied, because $\sigma_{new}\theta = (\sigma\{t/v\})\theta = \sigma(\{t/v\}\theta) = \sigma\theta = \theta$ (where the second last equality is from Lemma 3.12 and the last one holds by assumption).

Since Lemma 3.11 assures that the algorithm cannot run forever, we will eventually find a σ such that the while loop breaks ($N\sigma$ is a singleton). Hence the algorithm will finish with a unifier σ. But since θ could be any unifier for

N, the invariant gives that σ is a most general unifier. (It satisfies the equality $\theta = \sigma\lambda$ for any unifier θ and some λ. In fact it will always satisfy the stronger condition $\theta = \sigma\theta$ for any unifier θ.) □

Now there's only one thing left to verify, namely that

Lemma 3.15. *The algorithm returns* `fail` *if presented an ununifiable set.*

Proof. We know that the condition of the while loop (that $N\sigma$ is singleton) can only be fulfilled if there exists a unifier σ. And Lemma 3.11 assures that the while loop cannot run eternally. Hence the only possibility is that the algorithm breaks and returns the `fail` statement. □

3.6 Resolution

First-order resolution is a procedure for normal form formulas. We will therefore adopt the convention that *formula* is always understood to mean *normal form formula* (with quantifiers dropped) for the rest of this section.

3.6.1 The Resolution Rule.

In the propositional version, a resolvent R' of the (propositional) clauses P' and Q', with respect to some predicate letter B, was simply

$$R' = (P' - [B]) \cup (Q' - [\neg B])$$

Similarly, R is a *first-order resolvent* of the (first-order) clauses P and Q if, for some substitutions α, β,

$$R = (P\alpha - [A]) \cup (Q\beta - [\neg A])$$

where A is in $P\alpha$ and $\neg A$ in $Q\beta$.

To be able to fully utilize the unification algorithm, it will be necessary to express this rule in a different, and slightly restricted, way. First of all, we cannot depend on two different substitutions α and β; the same job has to be done by only one substitution θ. To not lose any deductive power, the variables of P and Q must be separate. The following *standard substitutions* are used to separate variables.

Definition. If $v_1, \ldots, v_n$ are the variables of a formula S (in, say, lexical order), then $\xi_S = \{x_1/v_1, \ldots, x_n/v_n\}$ and $\eta_S = \{y_1/v_1, \ldots, y_n/v_n\}$.

This means that we can express the resolution rule as

$$\begin{aligned} R &= (P\xi_P\alpha' - [A]) \cup (Q\eta_Q\beta' - [\neg A]) \\ &= (P\xi_P\theta - [A]) \cup (Q\eta_Q\theta - [\neg A]) \end{aligned}$$

where $\alpha = \xi_P\alpha'$, $\beta = \eta_Q\beta'$ and $\theta = \alpha' \cup \beta'$ (θ is a well-defined substitution, since α' and β' operates on disjoint sets of variables).

Second, we wish to move out θ, i.e. express the resolution rule as

$$\begin{aligned} R &= (P\xi_P\theta - [A]) \cup (Q\eta_Q\theta - [\neg A]) \\ &= (P\xi_P - L)\theta \cup (Q\eta_Q - M)\theta \end{aligned}$$

where $L \subseteq P\xi_P$ and $M \subseteq Q\eta_Q$, such that $L\theta = [A]$, $M\theta = [\neg A]$. Choosing $L = \{B \in P\xi_P : B\theta = A\}$ and $M = \{B \in Q\eta_Q : B\theta = \neg A\}$ should make it clear that it is possible find such subsets L and M.

Finally, in the procedure we will wish to start by picking a subset $L \subseteq P\xi_P$ containing only positive atomic formulas, and a subset $M \subseteq Q\eta_Q$ containing only negative atomic formulas; and then run the unification algorithm on $N = L \cup M'$ (where M' is like M but with negation signs removed). The criteria for the subsets are formalized by *key triples*.

Definition. A *key triple* for an ordered pair of clauses P and Q is a triple $\langle L, M, N\rangle$ such that

- L is a non-empty subset of $P\xi_P$ containing unnegated atomic formulas with the same predicate letter A,
- M is a non-empty subset of $Q\eta_Q$ containing negated atomic formulas with the predicate letter A,
- N is $L \cup M'$, where M' contains the formulas of M without negation sign, and,
- N should be unifiable by some most general unifier σ_N.

This gives us our final, procedural, way of describing a resolvent. R is a resolvent of P and Q, if there is a key triple $\langle L, M, N\rangle$ for P and Q such that

$$\begin{aligned} R &= (P\xi_P\sigma_N - L\sigma_N) \cup (Q\eta_Q\sigma_N - M\sigma_N) \\ &= (P\xi_P - L_{max})\sigma_N \cup (Q\eta_Q - M_{max})\sigma_N \end{aligned}$$

where $L_{max} = \{B \in P : B\sigma_N \in L\sigma_N\}$ and $M_{max} = \{B \in Q : B\sigma_N \in M\sigma_N\}$. (Note that $L\sigma_N = [A]$ and $M\sigma_N = [\neg A]$ for some atomic formula A.)

This last version is the version we will be using from here on. It is slightly restricted compared to the version we started out with, but as we shall see it is still powerful enough to build a complete proof procedure on. In the following, it is this last version we will mean by resolution/resolvent.

Example. Consider the clauses $P = [A(f(x)) \vee B(x)]$ and $Q = [\neg A(x) \vee \neg A(y)]$. Then $P\xi_P = [A(f(x_1)) \vee B(x_1)]$ and $Q\eta_Q = [\neg A(y_1) \vee \neg A(y_2)]$.

An available key triple $\langle L, M, N\rangle$ is this:

- $L = \{A(f(x_1))\} \subseteq P\xi_P$
- $M = \{\neg A(y_1), \neg A(y_2)\} \subseteq Q\eta_Q$
- $N = L \cup M' = \{A(f(x_1)), A(y_1), A(y_2)\}$

A most general unifier for N is $\sigma_N = \{f(x_1)/y_1, f(x_1)/y_2\}$. Thereby a resolvent for P and Q is:

$$\begin{aligned} R &= (P\xi_P\sigma_N - L\sigma_N) \cup (Q\eta_Q\sigma_N - M\sigma_N) \\ &= ([A(f(x_1)) \vee B(x_1)] - [A(f(x_1))]) \cup ([\neg A(f(x_1))] - [\neg A(f(x_1))]) \\ &= [B(x_1)] \end{aligned}$$

Theorem 3.16 (Soundness)**.** *If R is a resolvent of P and Q, then any model satisfying both P and Q will also satisfy R.*

Proof. Assume that R is a resolvent of P and Q with respect to some key triple $\langle L, M, N\rangle$, i.e.

$$R = (P\xi_P\sigma_N - L\sigma_N) \cup (Q\eta_Q\sigma_N - M\sigma_N)$$

and assume that $M = \langle H, I\rangle$ is a model satisfying both P and Q.

Then M will also satisfy $P\xi_P\sigma_N$ and $Q\eta_Q\sigma_N$. Now, let A be the single formula of $L\sigma_N$, and $\neg A$ the single formula of $M\sigma_M$.

Then we can write

$$\begin{aligned} P\xi_P\sigma_N &= [C_1 \vee \cdots \vee C_n \vee A] \\ Q\eta_Q\sigma_N &= [D_1 \vee \cdots \vee D_m \vee \neg A] \\ R &= [C_1 \vee \cdots \vee C_n \vee D_1 \vee \cdots \vee D_m] \end{aligned}$$

for some atomic formulas $C_1, \ldots, C_n$ and $D_1, \ldots, D_m$.

Now, just as in the propositional case, M will satisfy exactly one of A and $\neg A$ (any free variables in A consider universally quantified). In case M satisfies A, then M satisfies at least one of $D_1, \ldots, D_m$. And in case M satisfies $\neg A$, then M satisfies at least one of $C_1 \ldots, C_n$. Hence M satisfies R, which was to be proven. □

3.6.2 The $\mathcal{R}$ Operation

Just as in the propositional case; if S is a formula, then $\mathcal{R}(S)$ is S extended with all resolvents to clauses of S.

To show that the successive application of the $\mathcal{R}$ rule yields a complete proof procedure (i.e. that any the successive application of $\mathcal{R}$ to an unsatisfiable formula S, sooner or later will yield the empty clause), we will need the following lemmas.

Lemma 3.17. *For any formula S and Herbrand domain D_S, the $\mathcal{R}$ operation applied to the Herbrand expansion is included in the Herbrand expansion of $\mathcal{R}(S)$. Formally*

$$\mathcal{R}(\mathcal{E}(S, D_S)) \subseteq \mathcal{E}(\mathcal{R}(S), D_S)$$

Proof. Assume that R is in $\mathcal{R}(\mathcal{E}(S, D_S))$. Then, if $R \in \mathcal{E}(S, D_S)$, then R is also in $\mathcal{R}(\mathcal{E}(S, D_S))$, since the latter is a superset of the former.

Hence, suppose that R is not in $\mathcal{E}(S, D_S)$, but is a resolvent of two clauses $P\xi_P\alpha$ and $Q\eta_Q\beta$ in $\mathcal{E}(S, D_S)$ (where α and β are D_S-instantiations of $P\xi_P$ and $Q\eta_Q$ respectively, and P and Q are clauses of S.

Now, to show that R is also in $\mathcal{E}(\mathcal{R}(S), D_S)$, we should show that R is a D_S-instantiation of a resolvent T from two clauses in S. In fact, the two clauses are P and Q, as shall be apparent.

R is a resolvent of $P\xi_P\alpha$ and $Q\eta_Q\beta$. Hence, for some atomic formula A

$$R = (P\xi_P\alpha - [A]) \cup (Q\eta_Q\beta - [\neg A])$$

Since α and β operate on disjoint sets of variables (α on $x_1, x_2, \ldots$, and β on $y_1, y_2, \ldots$), we can let $\theta = \alpha \cup \beta$ and write

$$\begin{aligned} R &= (P\xi_P\theta - [A]) \cup (Q\eta_Q\theta - [\neg A]) \\ &= (P\xi_P - L)\theta \cup (Q\eta_Q - M)\theta \end{aligned}$$

(where L and M are subsets of $P\xi_P$ and $Q\eta_Q$ respectively).

So θ unifies L and M, and hence also $N = L \cup M'$ (where M' is like M, but with negation signs removed from its components). This means that N is also unified by a most general unifier σ_N (see unification above), and that $\theta = \sigma_N\lambda$ for some D_S-instantiation λ.

Thereby we have motivated that $\langle L, M, N\rangle$ is a key triple for P and Q. Hence

$$T = (P\xi_P - L)\sigma_N \cup (Q\eta_Q - M)\sigma_N$$

is a resolvent of P and Q.

And it is also apparent that R is a D_S-instantiation of T, namely T instantiated with λ, for

$$\begin{aligned} R &= (P\xi_P - L_{max})\theta \cup (Q\eta_Q - M_{max})\theta \\ &= ((P\xi_P - L_{max}) \cup (Q\eta_Q - M_{max}))\theta \\ &= ((P\xi_P - L_{max}) \cup (Q\eta_Q - M_{max}))\sigma_N\lambda \\ &= T\lambda \end{aligned}$$

Which is what we wanted. □

Induction yields that $\mathcal{R}^n(\mathcal{E}(S, D_S)) \subseteq \mathcal{E}(\mathcal{R}^n(S), D_S)$. Intuitively we can just "move in" the n $\mathcal{R}$ operators one by one. Hence

Lemma 3.18. $\mathcal{R}^n(\mathcal{E}(S, D_S)) \subseteq \mathcal{E}(\mathcal{R}^n(S), D_S)$.

3.6.3 The Procedure

Just as in the propositional case we keep applying the $\mathcal{R}$ operation to the given formula S until either $\mathcal{R}^n(S)$ contains the empty clause, or $\mathcal{R}^n(S) = \mathcal{R}^{n+1}(S)$ and $\mathcal{R}^n(S)$ does not contain the empty clause (Algorithm 4).

Algorithm 4 The-Resolution-Procedure(S)

```
i ← 0
loop
  i++
  if R^i(S) = R^{i-1}(S) then
    return satisfiable
  else if R^i(S) contains the empty clause then
    return unsatisfiable
  end if
end loop
```

The completeness theorem 3.20 below shows that `unsatisfiable` will be returned eventually if S is unsatisfiable, and the soundness theorem 3.16 above shows that `unsatisfiable` can be returned *only if* S is unsatisfiable.

It is also clear that `satisfiable` is returned only if we are completely stuck, with no hope ever finding the empty clause.

Hence we can confidently let the resolution procedure run and know that we can trust the returned answer.

Unfortunately we cannot know that it will *always* return `satisfiable` for satisfiable formulas. Turing's theorem 1.1 renders any theorem proving procedure thus powerful impossible.

Lemma 3.19 (Grounded Completeness)**.** *The resolution procedure applied to a grounded, unsatisfiable, normal form formula S will find the empty clause in finite time.*

Proof. Reading each atomic formula as a propositional letter (see proposition 3.7), the resolution rule becomes the same as the propositional resolution rule (substitutions are not available on grounded formulas).

Hence completeness theorem 2.4 for propositional logic applies. □

Theorem 3.20 (Completeness)**.** *$\mathcal{R}^n(S)$ will contain the empty clause for some n, if S is unsatisfiable.*

Proof. We know from corollary 3.6 that S is unsatisfiable if and only if there is a Herbrand domain D_S for S such that $\mathcal{E}(S, D_S)$ is unsatisfiable.

And due to lemma 3.19 the resolution procedure applied to the grounded set $\mathcal{E}(S, D_S)$ will eventually yield the empty clause (i.e. that $[] \in \mathcal{R}^n(\mathcal{E}(S, D_S))$ for some n), if $\mathcal{E}(S, D_S)$ is unsatisfiable.

But we also have from lemma 3.18 that for any Herbrand domain D_S and natural number n, then $\mathcal{R}^n(\mathcal{E}(S, D_S)) \subseteq \mathcal{E}(\mathcal{R}^n(S), D_S)$.

So if S is unsatisfiable, then $\mathcal{R}^n(\mathcal{E}(S, D_S))$ will contain the empty clause (for some D_S), and thereby also $[] \in \mathcal{E}(\mathcal{R}^n(S), D_S)$. But a Herbrand expansion of $\mathcal{R}^n(S)$ will never contain the empty clause if not $\mathcal{R}^n(S)$ does. Hence $R^n(S)$ must contain the empty clause. □

This theorem does not, of course, say anything about the size of n when the empty clause will be found. It only states that if the input is an unsatisfiable formula, the empty clause will be found *eventually*. There is no guarantee that n will not grow absurdly large before the empty clause is found.

In fact, n will sometimes grow absurdly large. For there can be no *computable function* $f : L \to \mathbb{N}$, such that f forms an upper bound for n. This is a rather strong statement, considering that for example

$$f(x) = x^{100x^{100x}}$$

is a perfectly computable function. The reason is that if there was one such function f, we could, for any input formula S, let the procedure go through the loop $f(S)$ times. If the empty clause still was not found, we would know that it would never be found. We would have had a (terribly inefficient) decision procedure for the satisfiability problem of first-order logic. But according to Turings theorem 1.1, no such procedure can exist.

Therefore any proof procedure for first-order logic is doomed to consume a terrible amount on time on at least *some* inputs. Paradoxically, this does not stop this algorithm from being rather quick on many inputs, as we shall see in the next section.

4 Applications

A natural question to raise is whether the theory developed so far is usable at all. Can it actually be mathematically useful?

In order to answer that we will no longer depend on theoretical arguments. Instead we will need an implementation of the resolution principle; that is an actual computer program, runnable on a normal computer, making use of the theory we have developed.

Fortunately, several such programs have already been developed. In fact, every year a competition is held between the best automated theorem provers, called CADE - Conference on Automated Deduction[6]. We shall have a closer look at a theorem prover by the name Prover9.

4.1 Prover9

Prover9[8] is a freely available (open source) automated theorem proving implementation, building on the resolution principle.

The resolution principle is extended with a couple of extra rules. Some of them are variations on the resolution rule (unit resolution, hyper resolution e.g.), others are rules introduced to handle equality (paramodulation, rewriting, flip). For some of them, short explanations are provided when they occur in proofs below.

Now lets have a look at what using Prover9 can look like.

4.1.1 Syllogism

Let us first see if Prover9 can prove the famous syllogism of Aristotle:

1. All men are mortal.
2. Socrates is a man.
3. Hence, Socrates is mortal.

Prover9 wants to be fed a set of assumptions (a natural choice would be item 1 and 2), and a set of goals (item 3 in this case), in a text file. It can look something like this.

```
formulas(assumptions).
  all x (man(x) -> mortal(x)).
  man(socrates).
end_of_list.

formulas(goals).
  mortal(socrates).
end_of_list.
```

Prover9 will then generate the following output.

```
============================== PROOF =================================

% Proof 1 at 0.02 (+ 0.00) seconds.
% Length of proof is 7.
% Level of proof is 3.
% Maximum clause weight is 0.
% Given clauses 0.

1 (all x (man(x) -> mortal(x))) # label(non_clause).  [assumption].
2 mortal(socrates) # label(non_clause) # label(goal).  [goal].
3 man(socrates).  [assumption].
4 -man(x) | mortal(x).  [clausify(1)].
5 mortal(socrates).  [resolve(3,a,4,a)].
6 -mortal(socrates).  [deny(2)].
7 $F.  [resolve(5,a,6,a)].

============================== end of proof ==========================
```

The first lines represents some general info about the proof: the time consumed, the length of the proof etc. This is an easy proof, and, accordingly, it went pretty fast.

The rest is the actual proof. We see that it first restates the given assumptions and the goal. In step 4 it then clausifies the "man so mortal" statement (i.e. it puts it on normal form), and then it finds the resolvent 5 from 3 and 4.

Combining 5 with the negation of the goal, it finds the empty clause (represented with $F).

4.1.2 Group Theory

Now let us consider some mathematically more interesting examples, namely two propositions of group theory.

A *group* is any structure satisfying the three axioms of group theory. The axioms are easily expressed as a list of assumptions in the Prover9 syntax.

```
formulas(assumptions).
  all x all y (x*y)*z=x*(y*z).
  all x ((x*e=e*x) & (e*x=x)).
  all x exists y ((x*y=e)   & (y*x=e)).
end_of_list.
```

Prover9 natively supports first-order logic with equality, so there is no need to supply with the axioms of equality. To handle equalities, it frequently uses a deduction rule called *paramodulation*. The paramodulation rule is most easily explained by an example: given the formulas $A(f(x))$ and $f(x) = y$, we can infer $A(y)$.

Formally, in its full generality, it becomes: For a formula $P(t)$ and an equality $s_1 = s_2$, such that for some substitutions σ and λ we have that $P(t)\sigma$ is $P(u)$ and $(s_1 = s_2)\lambda$ is $u = s_3$, we can infer $P(s_3)$.

Example 1. We wish to prove (from nothing but the group axioms), that every equation $a * x = b$ has a unique solution x. A by no means obvious consequence of the group axioms for anyone not familiar with the workings of group theory. The proposition can be formulated as the goal

```
formulas(goals).
  exists x (a*x=b & all y (a*y=b -> y=x)).
end_of_list.
```

Feeding Prover9 with the group axioms above and the goal, it quickly generates a proof. The proof is rather compact however, so we have used a utility called `prooftrans` (that comes bundled with Prover9), to expand it.

```
% Proof 1 at 0.02 (+ 0.01) seconds.
% Length of proof is 15.
% Level of proof is 4.
% Maximum clause weight is 11.
% Given clauses 15.

1 (all x all y (x * y) * z = x * (y * z)) # label(non_clause).  [assumption].
2 (all x (x * e = e * x & e * x = x)) # label(non_clause).  [assumption].
3 (all x exists y (x * y = e & y * x = e)) # label(non_clause).  [assumption].
4 (exists x (a * x = b & (all y (a * y = b -> y = x)))) # label(non_clause)
   # label(goal).  [goal].
5 (x * y) * z = x * (y * z).  [clausify(1)].
7 e * x = x.  [clausify(2)].
8 x * f1(x) = e.  [clausify(3)].
9 f1(x) * x = e.  [clausify(3)].
10 a * x != b | a * f2(x) = b.  [deny(4)].
11 a * x != b | f2(x) != x.  [deny(4)].
13A e * x = y * (f1(y) * x).  [para(8(a,1),5(a,1,1))].
13B x = y * (f1(y) * x).  [para(7(a,1),13A(a,1))].
13 x * (f1(x) * y) = y.  [copy(13B),flip(a)].
16A e * x = f1(y) * (y * x).  [para(9(a,1),5(a,1,1))].
16B x = f1(y) * (y * x).  [para(7(a,1),16A(a,1))].
16 f1(x) * (x * y) = y.  [copy(16B),flip(a)].
19 a * f2(f1(a) * b) = b.  [resolve(10,a,13,a)].
20 f2(f1(a) * b) != f1(a) * b.  [resolve(11,a,13,a)].
32A f1(a) * b = f2(f1(a) * b).  [para(19(a,1),16(a,1,2))].
32B f2(f1(a) * b) = f1(a) * b.  [copy(32A),flip(a)].
32 $F.  [resolve(20,a,32B,a)].
```

The first 11 steps are just conversion to normal form.

In 13A paramodulation is applied to sentences 5 and 8. 5 is instantiated to $(y * f1(y)) * x = y * (f1(y) * x)$ and 8 to $y * f1(y) = e$, which yields 13A: $e * x = y * (f1(y) * x)$.

13B is simply from the transitivity of equality, but it can also be expressed as an application of the paramodulation rule. To complete step 13, a flip makes it $x * (f1(x) * y) = y$ (Prover9 prefers to keep the "heavier" side of the equation to the left).

The rest of the steps are performed in similar ways to what have already

been discussed.

Example 2. Another theorem of group theory is that if all elements in a group G are their own inverses, G will be abelian (commutative). Starting with the same assumptions as in the last example (i.e. the group axioms), we wish the following goal to be proven.

```
formulas(goals).
all x x*x=e -> (all y all z y*z=z*y).
end_of_list.
```

This is not a problem for Prover9. The following proof (expanded with `prooftrans`) is generated almost instantaneously.

```
% Proof 1 at 0.02 (+ 0.00) seconds.
% Length of proof is 14.
% Level of proof is 5.
% Maximum clause weight is 11.
% Given clauses 12.

1 (all x all y a(a(x,y),z) = a(x,a(y,z))) # label(non_clause).  [assumption].
2 (all x (a(x,e) = a(e,x) & a(e,x) = x)) # label(non_clause).  [assumption].
4 (all x a(x,x) = e) -> (all y all z a(y,z) = a(z,y)) # label(non_clause)
  # label(goal).  [goal].
5 a(a(x,y),z) = a(x,a(y,z)).  [clausify(1)].
6 a(e,x) = a(x,e).  [clausify(2)].
7 a(e,x) = x.  [clausify(2)].
10 a(x,x) = e.  [deny(4)].
11 a(c2,c1) != a(c1,c2).  [deny(4)].
12A x = a(x,e).  [para(7(a,1),6(a,1))].
12 a(x,e) = x.  [copy(12A),flip(a)].
17A a(e,x) = a(y,a(y,x)).  [para(10(a,1),5(a,1,1))].
17B x = a(y,a(y,x)).  [para(7(a,1),17A(a,1))].
17 a(x,a(x,y)) = y.  [copy(17B),flip(a)].
18A e = a(x,a(y,a(x,y))).  [para(10(a,1),5(a,1))].
18 a(x,a(y,a(x,y))) = e.  [copy(18A),flip(a)].
24A a(x,e) = a(y,a(x,y)).  [para(18(a,1),17(a,1,2))].
24B x = a(y,a(x,y)).  [para(12(a,1),24A(a,1))].
24 a(x,a(y,x)) = y.  [copy(24B),flip(a)].
28 a(x,y) = a(y,x).  [para(24(a,1),17(a,1,2))].
29 $F.  [resolve(28,a,11,a)].
```

Again we see that the main part of the job is done by paramodulation.

4.2 Achievements

Perhaps not very surprisingly, abstract algebra is one of the areas where automated theorem proving has been most useful; proving a number of theorems previously unknown.

D.W. Loveland mentions a few of these in his article *Automated deduction: achievements and future directions*[4]. Another article exposing the power au-

tomated theorem proving applied to group theory is *Computer proofs in Group Theory*[9] by Yuan Yu.

The theory of automated theorem proving has also successfully been used in programming languages such as Prolog[7], which has a form of resolution built in. This allows the programmer to only specify a few logical relations, the actual proceedings of the calculations will be handled by the underlying automated theorem prover; ideally generating very comprehensive and easily maintainable source code.

References

[1] Stephen A. Cook, *The complexity of theorem-proving procedures* Proceedings of the Third Annual ACM Symposium on Theory of Computing, ACM, New York, 1971

[2] Martin Davis, Hilary Putnam, *A Computing Procedure for Quantification Theory*, Journal of the ACM (JACM), ACM, New York, 1960

[3] Melvin Fitting, *First-Order Logic and Automated Theorem Proving*, Springer, New York, 1996

[4] Donald W. Loveland, *Automated deduction: achievements and future directions*, Communications of the ACM, 2000

[5] Alan Robinson, *A Machine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM, 1965

[6] Geoff Sutcliffe, *The CADE-20 Automated Theorem Proving Competition* AI Communications, 2010

[7] *Prolog*, 16 April 2010, URL: http://www.swi-prolog.org/

[8] *Prover9*. 15 April 2010. URL: http://www.cs.unm.edu/~mccune/mace4/

[9] Yuan Yu, *Computer proofs in Group Theory*, Springer Netherlands, 2004