Raisonnement Automatisé: Principes et Applications (partie I: logique du premier ordre)

N. Peltier (CNRS, Université de Grenoble)

2009

This document contains the first part of the M2R course RAPA: *Automated Reasoning: Principle and Applications*. It presents the basis of first-order logic and automated deduction: syntax, semantics, transformation into clausal form, unification and the Resolution calculus (with selection functions and atom ordering). Some basic properties of the Resolution calculus are also investigated (w.r.t. complexity and termination).

This document is self-contained but additional references are provided for the interested reader. More details and additional explanations can be found in [5, 6]. [8] is an advanced textboook on the Resolution calculus and the Handbook of Automated Reasoning [12] covers the main lines of research in this field.

1 First Order Logic

First-order logic (FOL) is a formal language for expressing properties. Propositional logic allows one to express basic statements (s.t. "Paris is a town" or "Berlin is a town" or "Paris is the capital of France") and to combine them with logical connectives: $\neg$ (not), $\vee$ (or), $\wedge$ (and), $\Rightarrow$ (implies) and $\Leftrightarrow$ (equivalence). First-order logic extends this language by using predicate symbols and quantification over individuals. For instance, the property "to be a town" may be expressed by a predicate symbol *Town*, which can be applied to different individuals: $Town(Paris)$, $Town(Berlin)$, ... Using quantification, it is possible to express the property: "all countries have a capital": $\forall x[Country(x) \Rightarrow \exists y Capital(y, x)]$ (meaning: "for every x, if x is a country, then there exists a y such that y is the capital of x").

However, it is not possible in first order logic to express quantification over *sets* of individuals or over functions. For instance, the induction principle is not expressible in first-order logic: $\forall P[P(0) \wedge \forall x P(x) \Rightarrow P(succ(x))] \Rightarrow \forall x P(x)$ is *not* a sentence of FOL, due to the quantification over the sets of natural numbers P. Similarly, the property $\forall f \exists x f(x) = x$ (every function has a fixpoint) is not expressible in FOL.

1.1 Syntax

A *first-order language* $\mathcal{L}$ is a set containing:

- A set of *constant symbols*, usually denoted by $a, b, c, \ldots$
- A set of *function symbols* $f, g, h, \ldots$
- A set of *propositional variables* $P, Q, R, \ldots$
- A set of *predicate symbols*, also denoted by $P, Q, R, \ldots$

Each function or predicate symbol is associated to a unique natural number called its *arity* (number of arguments).

Throughout this document, we assume that a first-order language $\mathcal{L}$ is given, together with a set of *variables* $\mathcal{V}$, disjoint from the symbols in $\mathcal{L}$. Terms and formulae are defined relatively to these sets.

Definition 1 *(Terms) The set of* terms *is the* ***smallest*** *set that satisfies the following properties:*

- *Every constant symbol is a term.*
- *Every variable is a term.*
- *If $t_1, \ldots, t_n$ are terms and if f is a function symbol of arity n then $f(t_1, \ldots, t_n)$ is a term.*

We emphasize that this definition is *inductive* ("...smallest set that ..."). All the terms must be of one of the above forms and infinite terms of the form $f(f(f(\ldots)))$ are forbidden.

Alternatively, terms can be seen as *trees* labeled by symbols in $\mathcal{L}$.

Remark 2 *For the sake of uniformity, one can also view constant symbols as function symbols of arity* 0 *(nullary functions). If this convention is used, the first line in Definition 1 may be deleted. An expression of the form $f(t_1, \ldots, t_n)$ where $n = 0$ is to be read as the constant f.*

We denote by $Var(t)$ the set of variables occurring in the term t. This set is inductively defined as follows:

$$\begin{array}{lcll} Var(t) & = & \{t\} & \text{if } t \text{ is a variable} \\ Var(t) & = & \emptyset & \text{if } t \text{ is a constant symbol} \\ Var(t) & = & \bigcup_{i=1}^{n} Var(t_i) & \text{if } t \text{ is of the form } f(t_1, \ldots, t_n) \end{array}$$

First-order formulae are built inductively from terms using the predicate symbols and propositional variables in $\mathcal{L}$ and the logical symbols $\vee$ (or), $\wedge$ (and), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence), $\forall, \exists$ (quantification).

Definition 3 *(Formulae) The set of formulae is the smallest set that satisfies the following properties:*

- *Every propositional variable is a formula.*
- *"true" and "false" are formulae.*
- *If $t_1, \ldots, t_n$ are terms and P is a predicate symbol of arity n then $P(t_1, \ldots, t_n)$ is a formula.*
- *If ϕ is a formula then $(\neg\phi)$ is also a formula.*
- *If ϕ_1, ϕ_2 are formulae, then $(\phi_1 \wedge \phi_2)$, $(\phi_1 \vee \phi_2)$, $(\phi_1 \Rightarrow \phi_2)$ and $(\phi_1 \Rightarrow \phi_2)$ are formulae.*
- *If ϕ is a formula and x is a variable, then $(\forall x\phi)$ and $(\exists x\phi)$ are formulae.*

In the above definition, formulae are always written with parentheses. In practice, some parentheses may of course be omitted, and the usual priority rules are used to reconstruct the corresponding formula. The priority rank is as follows: $\forall, \exists > \neg > \wedge > \vee > \Rightarrow$. For instance $\neg P \wedge Q \vee R \Rightarrow P$ should be read as $(((\neg P) \wedge Q) \vee R) \Rightarrow R$.

A formula ψ is said to be a *subformula* of ϕ if ψ is either ϕ or a formula occurring inside ϕ (the formal definition is left to the reader).

Formulae that contain no logical symbols (i.e. that are propositional variables or of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol) are called *atoms* (or *atomic formulae*). A formula that is either an atomic formula or the negation of a atomic formula is called a *literal*.

For instance, the formula $(P \vee (\forall x Q(f(x)))) \Rightarrow (\exists x \neg R(x, x))$ contains 3 atoms: $P, Q(f(x))$ and $R(x, x)$. All these atoms are literals, as well as $\neg R(x, x)$. We say that a variable x is *free* in a formula ϕ if it occurs in ϕ, but not on the scope of a quantifier $\forall x$ or $\exists x$. Formally, we denote by $FVar(\phi)$ the set of free variables of ϕ, defined as follows:

$FVar(\phi)$	$=$	$\emptyset$	If ϕ is equal to *true* or *false*
$FVar(\phi)$	$=$	$\emptyset$	if ϕ is a propositional variable
$FVar(\phi)$	$=$	$\bigcup_{i=1}^{n} Var(t_i)$	if ϕ is of the form $P(t_1, \ldots, t_n)$
$FVar(\phi)$	$=$	$FVar(\psi)$	if ϕ is $\neg\psi$
$FVar(\phi)$	$=$	$FVar(\psi_1) \cup FVar(\psi_2)$	if ϕ is $\psi_1 \star \psi_2$ with $\star \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$
$FVar(\phi)$	$=$	$FVar(\psi) \setminus \{x\}$	if ϕ is $\exists x\psi$ or $\forall x\phi$

A formula is said to be *closed* if it has no free variable. A variable occurring on the scope of a quantifier is said to be *bound*. Note that the same variable may be free and bound simultaneously, for instance x in $p(x) \vee \forall x\ p(x)$. In this case, the two occurrences of x denote different objects (the first occurrence of x is a free variable, whose value is unknown, whereas the second one ranges over the whole domain). In practice, to avoid confusion, the variables should be renamed: $p(x) \vee \forall y\ p(y)$.

1.2 Substitutions

A *substitution* is a function mapping every variable to a term. The domain of a substitution σ is the set of variables x s.t. $\sigma(x) \neq x$ (usually the domain is assumed to be finite). A substitution σ of domain $x_1, \ldots, x_n$ is denoted as follows: $\{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$.

A substitution is said to be a *renaming* if for every variable x, $\sigma(x)$ is a variable, and if σ is injective.

If σ and θ are substitutions of disjoint domains, then $\sigma \cup \theta$ denotes the union of σ and θ: $(\sigma \cup \theta)(x) \stackrel{\text{def}}{=} \sigma(x)$ if $x \in dom(\sigma)$ and $(\sigma \cup \theta)(x) \stackrel{\text{def}}{=} \theta(x)$ if $x \in dom(\theta)$.

The image of a term t by a substitution σ is obtained by replacing any variable x occurring in t by $\sigma(x)$. It is usually denoted by $t\sigma$. Formally:

$$
\begin{array}{lcll}
a\sigma & \stackrel{\text{def}}{=} & a & \text{if } a \text{ is a constant symbol.} \\
x\sigma & \stackrel{\text{def}}{=} & \sigma(x) & \text{if } x \text{ is a variable.} \\
f(t_1, \ldots, t_n)\sigma & \stackrel{\text{def}}{=} & f(t_1\sigma, \ldots, t_n\sigma) &
\end{array}
$$

Substitutions can also be applied to formulae, exactly in the same way. However, a difficulty occurs when the formula contains quantifiers, because in this case there can be conflicts between the variables in the domain of the substitution and the bound variables occurring in the formula. The following example will clarify this. Assume that the substitution $\{x \mapsto a\}$ is applied to the formula $\forall x\ p(x)$. The variable x occurring in the formula should *not* be replaced by a. Indeed, since it occurs on the scope of a quantifier, it has no link to the variable x occurring in the substitution which is a free variable (although both variables have the same name they do not represent the same objects - from a programming point of view one could write that they do not occur in the same environment).

Thus, the formula should be renamed into (for example): $\forall y\ p(y)$ before the substitution can be applied. The result is $\forall y\ p(y)$.

Similarly, assume that a substitution $\{x \mapsto f(y)\}$ is applied to the formula $\forall x \exists y P(x, y)$. Once again, the variable y occurring in the substitution (more precisely in the term $f(y)$) is different from the bound variable y in the formula. The result should be: $\forall x \exists y' P(f(y), y')$ (and not $\forall x \exists y P(f(y), f(y))$ which has a different meaning).

Formally, we denote by $Var(\sigma)$ the set of variables occurring either in the domain of σ or in a term $t = \sigma(x)$, for some $x \in dom(\sigma)$. If the bound variables do not occur in $Var(\sigma)$ then the substitution is applied normally (as for terms), by replacing any variable x by $\sigma(x)$:

$$
\begin{array}{lcll}
p\sigma & \stackrel{\text{def}}{=} & p & \text{if } p \text{ is a propositional variable} \\
\phi\sigma & \stackrel{\text{def}}{=} & \phi & \text{if } \phi \text{ is equal to } \textit{true} \text{ or } \textit{false} \\
p(t_1,\ldots,t_n)\sigma & \stackrel{\text{def}}{=} & p(t_1\sigma,\ldots,t_n\sigma) & \\
(\neg\phi)\sigma & \stackrel{\text{def}}{=} & \neg(\phi\sigma) & \\
(\phi_1 \star \phi_2)\sigma & \stackrel{\text{def}}{=} & \phi_1\sigma \star \phi_2\sigma & \text{where } \star \text{ is either } \wedge, \vee, \Rightarrow \text{ or } \Leftrightarrow \\
(\exists x\phi)\sigma & \stackrel{\text{def}}{=} & (\exists x)\phi\sigma & \text{if } x \notin \mathit{Var}(\sigma) \\
(\forall x\phi)\sigma & \stackrel{\text{def}}{=} & (\forall x)\phi\sigma & \text{if } x \notin \mathit{Var}(\sigma)
\end{array}
$$

If a bound variable occurs in $\mathit{Var}(\sigma)$, then it should be renamed before applying σ, as shown that the following definitions:

$(\exists x\phi)\sigma \stackrel{\text{def}}{=} (\exists x')\phi\{x \mapsto x'\}\sigma$
if $x \in \mathit{Var}(\sigma)$ and x' is a new variable not occurring in ϕ or in $\mathit{Var}(\sigma)$
$(\forall x\phi)\sigma \stackrel{\text{def}}{=} (\forall x')\phi\{x \mapsto x'\}\sigma$
if $x \in \mathit{Var}(\sigma)$ and x' is a new variable not occurring in ϕ or in $\mathit{Var}(\sigma)$

If t is a term (or formula) and σ is a substitution, then $t\sigma$ is called an *instance* of t.

If σ, θ are two substitutions, then $\sigma\theta$ denotes the composition of σ and θ (σ is applied first). If $\gamma = \sigma\theta$ then σ is said to be *more general* than γ and γ is said to be an *instance* of σ.

1.3 Semantics

Semantics associate a truth value (true or false) to a given formula. It is of course impossible in general to associate a unique truth value to a given formula because this value depends on the meaning of the non logical symbols occurring in the formula.

For instance, the formula $P \wedge \neg P$ should obviously have the truth value *false*, regardless of the meaning of P because P cannot be false and true simultaneously (in boolean algebra $P.\overline{P}$ is 0). Similarly, $Q(a) \Rightarrow Q(a)$ should be *true*. But what is the meaning of the formula: $\exists x f(x) = x$? Obviously it depends on f. If for instance f is the successor function $x \mapsto x + 1$ then the formula is *false* (there is no x that is its own successor), but if f is $x \mapsto 2 \times x$, then the formula should be *true* ($x = 0$).

Thus, before affecting a truth value to a formula, it is necessary to specify the meaning of the symbols in $\mathcal{L}$ (the meaning of the logical symbols $\vee, \wedge, \Rightarrow, \Leftrightarrow, \forall, \exists$ is fixed). This is what we call an *interpretation*.

More formally:

Definition 4 *(Interpretation)*
An interpretation *is defined by a* domain D_I *that is a* ***non empty*** *set and by a function mapping:*

- *Each constant symbol a to an element $a_I \in D_I$.*
- *Each function symbol f of arity n to a function f_I from D_I^n into D_I.*
- *Each propositional variable P to a truth value P_I (P_I is either true or false).*
- *Each predicate symbol P of arity n to a function P_I from D_I^n into $\{true, false\}$ (since this function has only two possible values, one could simply denote it by the set of the tuples that are associated to true).*
- *Each variable x to an element x_I of D_I.*

Once the meaning of the constant and function symbols and of the variables are known, it is easy to define the value of all terms, by induction:

Definition 5 *(Value of a Term) If t is a term and I is an interpretation then $[t]_I$ (the value of the term t in I) is inductively defined as follows:*

- *If t is a constant symbol a then $[t]_I \stackrel{def}{=} a_I$.*
- *If t is a variable x then $[t]_I \stackrel{def}{=} x_I$.*
- *If t is of the form $f(t_1, \ldots, t_n)$ then $[t]_I \stackrel{def}{=} f_I([t_1]_I, \ldots, [t_n]_I)$.*

Note that the value of a term is always an element of the domain, by definition.

Once the range of the variables (domain) and the meaning of terms are known, it is easy to define the truth value of a formula.

We need to introduce a notation: if I is an interpretation, then the expression $I\{x \leftarrow v\}$ (where x is a variable or a constant and v is an element of the domain D_I of I) denotes an interpretation J which is identical to I (same domain and same interpretation of all symbols), except for the interpretation of x that is defined as follows: $x_J \stackrel{\text{def}}{=} v$.

Definition 6 *(Value of a Formula) If ϕ is a formula and I is an interpretation, then $[\phi]_I$ (the truth value of ϕ in I) is defined as follows:*

- *If ϕ is a propositional variable P, then $[\phi]_I \stackrel{def}{=} P_I$.*
- *If ϕ is of the form $P(t_1, \ldots, t_n)$ then $[\phi]_I \stackrel{def}{=} P_I([t_1]_I, \ldots, [t_n]_I)$.*
- *If ϕ is of the form $\neg\psi$ then* $[\phi]_I \stackrel{def}{=} \begin{cases} true & \text{if } [\psi]_I = false \\ false & \text{otherwise} \end{cases}$
- *If ϕ is of the form $\psi_1 \wedge \psi_2$ then:*

  $[\phi]_I \stackrel{def}{=} \begin{cases} true & \text{if } [\psi_1]_I = true \textbf{ and } [\psi_2]_I = true \\ false & \text{otherwise} \end{cases}$

- *If ϕ is of the form $\psi_1 \vee \psi_2$ then:*

  $$[\phi]_I \stackrel{def}{=} \begin{cases} true & \text{if } [\psi_1]_I = true \textbf{ or } [\psi_2]_I = true \\ false & \text{otherwise} \end{cases}$$

- *If ϕ is of the form $\psi_1 \Rightarrow \psi_2$ then:*

  $$[\phi]_I \stackrel{def}{=} \begin{cases} true & \text{if } [\psi_1]_I = false \textbf{ or } [\psi_2]_I = true \\ false & \text{otherwise} \end{cases}$$

  This means that $\psi_1 \Rightarrow \psi_2$ is equivalent to $\neg\psi_1 \vee \psi_2$.

- *If ϕ is of the form $\psi_1 \Leftrightarrow \psi_2$ then:*

  $$[\phi]_I \stackrel{def}{=} \begin{cases} true & \text{if } [\psi_1]_I = [\psi_2]_I \\ false & \text{otherwise} \end{cases}$$

- *If ϕ is of the form $\forall x\ \psi$ then:*

  $$[\phi]_I \stackrel{def}{=} \begin{cases} true & \text{if for any element } v \in D_I,\ [\psi]_{I\{x \leftarrow v\}} = true \\ false & \text{otherwise} \end{cases}$$

  Essentially, ϕ is true iff ψ is true regardless of the value of x.

- *If ϕ is of the form $\exists x\ \psi$ then:*

  $$[\phi]_I \stackrel{def}{=} \begin{cases} true & \text{if there exists } v \in D_I \text{ such that } [\psi]_{I\{x \leftarrow v\}} = true \\ false & \text{otherwise} \end{cases}$$

Notice that for the sake of simplicity we assume that I gives a value to all constant/function/predicate symbols in the language and to all variables. But, obviously, the truth value of a formula ϕ in an interpretation I depends only on the value of the symbols and of the variables (freely) occurring in ϕ (the values of the remaining symbols are irrelevant). In particular, the value of a closed formula does not depend on variables.

We write $I \models \phi$ (or I *validates* ϕ) iff $[\forall x_1 \ldots \forall x_n \phi]_I = true$, where $\{x_1, \ldots, x_n\}$ is the set of the free variables in ϕ. Then I is called a *model* of ϕ. Notice that the free variables are universally quantified. This implies that the interpretation of the variables in the model is irrelevant (the formula must be true for all possible values of the variables).

If S is a set of formulae, then we write $I \models S$ iff for all $\phi \in S$, $I \models \phi$.

A formula (or set of formulae) ϕ is said to be *satisfiable* if it has a model, unsatisfiable otherwise. It is said to be *valid* if every interpretation is a model. Obviously, ϕ is valid iff $\neg\phi$ is unsatisfiable. Two formulae ϕ, ψ are said to be *equivalent* (written $\phi \equiv \psi$) iff for any interpretation I, $[\phi]_I = [\psi]_I$ (i.e. if $I \models (\phi \Leftrightarrow \psi)$). A formula ψ is said to be a *logical consequence* of ϕ iff for any interpretation I s.t. $[\phi]_I = true$ we have $[\psi]_I = true$. This is written $\phi \models \psi$ Beware that the same notation denotes two distinct relations: "the formula ψ is a logical consequence of ϕ" is denoted by $\phi \models \psi$ and "the interpretation I is a model of ϕ" by $I \models \phi$.

Two important points deserve to be emphasized because they are often misunderstood:

Remark 7 *If I is an interpretation and ϕ_1, ϕ_2 are formulae, then the fact that $I \models \phi_1 \vee \phi_2$ does* ***not*** *imply that there exists $i \in \{1,2\}$ s.t. $I \models \phi_i$, although we have $[\phi_1 \vee \phi_2]_I = true$ iff $[\phi_1]_I = true$ or $[\phi_2]_I = true$. Indeed, ϕ_1, ϕ_2 may contain free variables and the index i s.t. ϕ_i is true can depend on the interpretation of these variables. For instance if Even and Odd are interpreted as the usual predicates on natural numbers, then we have $I \models Even(x) \vee Odd(x)$ but $I \not\models Even(x)$ and $I \not\models Odd(x)$ (every natural number is either even or odd, but it is not true that all natural numbers are even, nor that all natural numbers are odd).*

Of course, if $\phi_1 \vee \phi_2$ is closed then the above property is true.

Remark 8 *Similarly, if ψ, ϕ_1, ϕ_2 are formulae, then we can have $\psi \models \phi_1 \vee \phi_2$ but $\psi \not\models \phi_1$ and $\psi \not\models \phi_2$, even if ϕ_1, ϕ_2 are closed. The reason is that $\psi \models \phi_1 \vee \phi_2$ means that for any model I of ψ there exists $i \in \{1,2\}$ s.t. $I \models \phi_i$, but the index i such that ϕ_i is true* can depend on *I. Assume for instance that ψ is true, that ϕ_1 is a propositional variable P and that $\phi_2 \stackrel{def}{=} \neg P$. Then $\psi \models P \vee \neg P$ but $\psi \not\models P$ and $\psi \not\models \neg P$.*

There are alternative (equivalent) ways of defining the semantics of a first-order formula.

Sometimes, the interpretation of the free variables is not included in the interpretation itself, but is given separately by a another function (often called a *valuation*). Our definition is more uniform.

In order to avoid having to assign values to variables, we could also replace quantified variables by new ("fresh") constant symbols before interpreting them. The value of an existential formula would then be defined as follows: $[\exists x \phi]_I = true$ iff there exists an element $v \in D_I$ s.t. $[\phi\{x \mapsto c\}]_{I\{c \leftarrow v\}} = true$, where c is a new constant symbol, not occurring in ϕ. The definition for universal quantifiers is similar.

All these definitions are of course equivalent (this can be shown by an easy induction on the formulae).

The following lemmata state easy consequences of the definition, namely the possibility of replacing, in a formula, a subformula by an equivalent one, without affecting the truth value. Similarly, a variable may be replaced by a term having the same value.

Lemma 9 *(Replacement Lemma) Let ϕ be a formula. Let ψ be a subformula occurring in ϕ. Let ψ' be a formula equivalent to ψ and let ϕ' be a formula obtained from ϕ by replacing the formula ψ by ψ'.*

ϕ is equivalent to ϕ'.

Proof The proof is by an easy induction on the size of the formula. It is left to the reader as an exercise. ■

Lemma 10 *Let I be an interpretation, x be a variable, t be a term and ϕ be a formula (or term).*

If $x_I = [t]_I$ then $[\phi]_I = [\phi\{x \mapsto t\}]_I$.

Proof This is an immediate consequence of the definition (the detailed proof is by an easy induction on the size of ϕ). ■

Corollary 11 *Let ϕ be a formula, x be a variable and t be a term.*
The formulae $(\forall x\ \phi) \Rightarrow \phi\{x \mapsto t\}$ and $\phi\{x \mapsto t\} \Rightarrow (\exists x\ \phi)$ are valid.

Proof We only give he proof for the first formula. Let I be an interpretation. We have to show that $[\forall x\phi \Rightarrow \phi\{x \mapsto t\}]_I = \mathit{true}$. We assume, w.l.o.g. that x does not occur in t (if it is the case then we simply rename the formula $\forall x\phi$ into the equivalent formula $\forall x'\phi\{x \mapsto x'\}$ where x' occurs neither in ϕ not in t).

By definition $[\forall x\phi \Rightarrow \phi\{x \mapsto t\}]_I = \mathit{true}$ iff either $[\forall x\phi]_I = \mathit{false}$ or $[\phi\{x \mapsto t\}]_I = \mathit{true}$. Thus we assume that $[\forall x\phi]_I = \mathit{true}$. Let $v = [t]_I$ and let $J = I\{x \leftarrow v\}$. By definition of the interpretation of $\forall x\phi$ we have $[\phi]_J = \mathit{true}$. By definition of J we have $x_J = [t]_I = [t]_J$ (since x does not occur in t, the value of t is the same in I and in J).

By Lemma 10 we deduce that $[\phi\{x \mapsto t\}]_J = \mathit{true}$. But since $\phi\{x \mapsto t\}$ does not contain x, we have $[\phi\{x \mapsto t\}]_I = [\phi\{x \mapsto t\}]_J = \mathit{true}$. ■

2 Clausal Normal Form

2.1 Clauses

Definition 12 *A* clause *is a formula of the form $\bigvee_{i=1}^{n} L_i$ where the $L_1, \ldots, L_n$ are literals (i.e. atoms or negations of atoms).*

We may have $n = 0$, then the clause is empty. The empty clause is denoted by $\Box$. By convention, it is equivalent to *false* (empty disjunction).

Alternative notations are commonly used to denote clauses in the literature. A clause is often considered as a set (or multiset) of literals (the disjunction is then implicit). This is possible since disjunction is commutative and associative. Thus the meaning of a clause does not depend on the order of the literals.

A clause may be also considered as an implication, by regrouping the negative literals before an implication sign: $\neg\phi_1 \vee \ldots \vee \neg\phi_n \vee \psi_1 \vee \ldots \vee \psi_m$ is equivalent to $\phi_1 \wedge \ldots \phi_n \Rightarrow \psi_1 \vee \ldots \vee \psi_m$. For instance the clause $\neg P(x) \vee Q(x, y) \vee \neg R(y)$ may be written $\{\neg P(x), Q(x, y), \neg R(y)\}$ or $P(x) \wedge R(y) \Rightarrow Q(x, y)$. In Prolog notation, this is written: $Q(x, y) : -P(x), R(y)$.

2.2 Transformation into Clausal Form

It is possible to transform any formula ϕ into a set (conjunction) of clauses S that is sat-equivalent to ϕ, i.e. such that ϕ is satisfiable iff S is satisfiable. S is called a *clausal form* of ϕ (the clausal form is non unique).

In order to transform formulae into sets of clauses, one has to:

1. Eliminate all occurrences of the logical symbols $\Rightarrow$ and $\Leftrightarrow$. Fortunately these symbols can easily by expressed using $\vee$ and $\wedge$ only.

2. Put all negation symbols before atoms. Negations of complex formulae must be eliminated. This is possible by De Morgan's laws which transform negations of conjunctions into disjunctions of negations and conversely.

3. Eliminate existential quantifiers. This is the most difficult part and the only step that does not preserve equivalence.

4. Apply the distributivity rule in order to obtain conjunctions of disjunctions and put all universal quantifiers before disjunctions. Then, universal quantifiers can be simply eliminated (since any free variable is already implicitly universally quantified).

We present an algorithm in order to construct a clausal form of any formula. This algorithm is defined as a set of transformation rules, operating on formulae. These rules are applied in an indeterministic way, on the considered formula or on its subformulae.

2.2.1 Equivalence Preserving Rules

The first set of rules eliminates all occurrences of $\Rightarrow$ and $\Leftrightarrow$ (step 1 above).

$$\begin{array}{lllll} CF_{\Rightarrow} & \phi \Rightarrow \psi & \rightarrow & (\neg\phi \vee \psi) \\ CF_{\Leftrightarrow} & \phi \Leftrightarrow \psi & \rightarrow & (\neg\phi \vee \psi) \wedge (\neg\psi \vee \phi) \end{array}$$

There is an other way of eliminating the symbol $\Leftrightarrow$:

$$CF'_{\Leftrightarrow} \quad \phi \Leftrightarrow \psi \quad \rightarrow \quad (\phi \wedge \psi) \vee (\neg\psi \wedge \neg\phi)$$

Theoretically these two rules are equivalent. The first one ($CF_{\Leftrightarrow}$) is more natural (since one want to eventually obtain conjunctions of disjunctions), but the second one ($CF'_{\Leftrightarrow}$) is more efficient if the symbol $\Leftrightarrow$ occurs in the scope of an odd number of negation symbols. For instance, consider the formula $\neg(P \Leftrightarrow Q)$. One get, by applying $CF'_{\Leftrightarrow}$, the formula: $\neg[(P \wedge Q) \vee (\neg P \wedge \neg Q)]$. By De Morgan's law (see below) this formula is transformed into: $(\neg P \vee \neg Q) \wedge (P \vee Q)$, which is in clausal form (it is a conjunction of clauses). The reader can easily check that applying $CF_{\Leftrightarrow}$ yields a more complex derivation.

The second set of rules removes the negations occurring behind a complex formula (step 2).

$$\begin{array}{llll} CF_{\neg\neg} & \neg\neg\phi & \rightarrow & \phi \\ CF_{\neg\wedge} & \neg(\phi \wedge \psi) & \rightarrow & (\neg\phi) \vee (\neg\psi) \\ CF_{\neg\vee} & \neg(\phi \vee \psi) & \rightarrow & (\neg\phi) \wedge (\neg\psi) \\ CF_{\neg\exists} & \neg(\exists x\phi) & \rightarrow & \forall x(\neg\phi) \\ CF_{\neg\forall} & \neg(\forall x\phi) & \rightarrow & \exists x(\neg\phi) \end{array}$$

Finally, the following rules apply the distributivity axiom and put all quantifiers behind disjunctions:

$$\begin{array}{lllll} CF_{\vee\wedge} & \phi \vee (\phi_1 \wedge \phi_2) & \rightarrow & (\phi \vee \phi_1) \wedge (\phi \vee \phi_2) \\ CF_{\forall\wedge} & \forall x(\phi \wedge \psi) & \rightarrow & (\forall x\phi) \wedge (\forall x\psi) \\ CF_{\vee\forall} & \phi \vee (\forall x\psi) & \rightarrow & (\forall y)(\phi \vee \psi\{x \rightarrow y\}) \end{array}$$

where y is either x if $x \notin FVar(\phi)$, or a new variable not occurring in ϕ, ψ, if $x \in Var(\phi)$.

These rules must be applied modulo the usual associative and commutative properties of $\vee, \wedge$. For instance $CF_{\vee\wedge}$ can be applied on $(P \wedge Q) \vee R$ yielding $(P \vee R) \wedge (Q \vee R)$.

In the rule $CF_{\vee\forall}$, renaming x into y in the case where x occurs in ϕ is *essential* in order to avoid conflicts on the variable names. Consider for instance the formula $(\forall xP(x)) \vee (\forall xQ(x))$. By the first application of the rule (on the subformula $(\forall xP(x))$ one get: $\forall x(P(x) \vee (\forall xQ(x)))$ (no renaming is necessary since x is not free in $\forall xQ(x)$). Afterwards, the rule is applied again, now with the subformula $(\forall xQ(x))$. Here, x freely occurs in $P(x)$, thus one has to rename x into a new variable y, yielding the clause: $\forall x, y(P(x) \vee Q(y))$.

If the renaming is not performed, then one gets $\forall x(P(x) \vee Q(x))$ which has a different meaning (incorrect in the sense that the semantics of the original formula are not preserved).

Lemma 13 *The rules $CF_{\Rightarrow}$, $CF_{\Leftrightarrow}$, $CF_{\neg\neg}$, $CF_{\neg\wedge}$, $CF_{\neg\vee}$, $CF_{\neg\exists}$, $CF_{\neg\forall}$, $CF_{\vee\wedge}$, $CF_{\vee\forall}$ and $CF_{\forall\wedge}$ preserve equivalence, i.e. if ϕ, ψ are two formulae and if ψ is obtained from ϕ by applying one of these rules (on any subformula in ϕ) then ψ and ϕ are equivalent.*

Proof The proof is a straightforward consequence of Definition 6 and of Lemma 9. One has to check in each case that the left-hand side is equivalent to the right-hand side. This is left as an exercise to the reader. ■

If a formula ϕ is irreducible by the rules for negation $CF_{\neg\neg}$, $CF_{\neg\wedge}$, $CF_{\neg\vee}$, $CF_{\neg\exists}$ and $CF_{\neg\forall}$ and by the rules $CF_{\Rightarrow}$ and $CF_{\Leftrightarrow}$ (i.e. if these rules cannot be applied on any subformula of ϕ) then ϕ is said in *negation normal form*. Intuitively, ϕ is built from literals using only the connectives $\vee, \wedge, \forall, \exists$.

2.2.2 Skolemisation

We now show how to eliminate existential quantifiers.

The skolemisation rule (from the Norvegian mathematician Thoralf Albert Skolem, 1887-1963) is a rule transforming existential formulae into sat-equivalent formulae without $\exists$. The idea is, given a formula $(\exists x\phi)$ to introduce a function f associating to the free variables $y_1, \ldots, y_n$ in $(\exists x\phi)$, the element x s.t. ϕ holds (if it exists). Then x can be simply replaced by $f(y_1, \ldots, y_n)$.

In the particular case where $(\exists x\phi)$ contains no free variables, then x is simply replaced by a constant a.

This idea is formalized by the following rule:

$$CF_{\exists} \quad \exists x\phi \quad \rightarrow \quad \phi\{x \mapsto f(y_1, \ldots, y_n)\}$$

Where:

- $y_1, \ldots, y_n$ is the set of variables that are free in $\exists x \phi$ (we may have $n = 0$, in this case $f(y_1, \ldots, y_n)$ is to be read as f).
- f is a new function symbol (or a constant symbol if $n = 0$) **not occurring in the whole formula**[1].
- $\exists x \phi$ does not occur on the scope of a symbol $\neg, \Rightarrow$ or $\Leftrightarrow$.

In order to prove that $CF_\exists$ preserves satisfiability, we need the following:

Lemma 14 *Let ϕ be a formula. Let ψ be a subformula occurring in ϕ but not on the scope of a connective $\neg, \Leftrightarrow$ or $\Rightarrow$. Let I be an interpretation and let ψ' be a formula s.t. $I \models \psi \Rightarrow \psi'$. Let ϕ' be the formula obtained by replacing ψ by ψ' in the formula ϕ.*

We have either $[\phi]_I = false$ or $[\phi']_I = true$.

Proof We reason by induction on the size of ϕ. By definition ϕ contains the formula ψ. One of the following conditions holds:

- ϕ is ψ. Then we must have $\phi' = \psi'$ and the property holds by definition since $I \models \psi \Rightarrow \psi'$.
- If ϕ is not ψ, then ϕ must be a complex formulae, containing ψ and its root symbol is either $\vee, \wedge$ or a quantification. We consider each case separately:

  $\vee$: ϕ is of the form $\gamma_1 \vee \gamma_2$, where one of the two formulae γ_1, γ_2 contains ψ. Assume, without loss of generality, that ψ occurs in γ_1. By definition ϕ is of the form $\gamma_1' \vee \gamma_2$ where γ_1' is obtained by replacing ψ by ψ' in γ_1.

  We assume that $[\phi]_I = true$ and we show that $[\phi']_I = true$. By definition of the semantics, there exists $i \in \{1, 2\}$ s.t. $[\gamma_i]_I = true$. If $i = 2$ then we have $[\phi']_I = true$ since $\phi' = \gamma_1' \vee \gamma_2$. Otherwise, we apply the induction hypothesis on γ_1, γ_1' (this is possible since γ_1' is strictly smaller than ϕ). Since $[\gamma_1]_I = true$, we deduce that $[\gamma_1']_I = true$, hence that $[\phi']_I = true$.

  $\wedge$: The proof is similar if ϕ is of the form $\gamma_1 \wedge \gamma_2$.

  $\exists$: ϕ is of the form $\exists x \gamma$, where γ contains ψ. By definition ϕ is of the form $\exists x \gamma'$ where γ' is obtained by replacing ψ by ψ' in γ.

  We assume that $[\phi]_I = true$ and we show that $[\phi']_I = true$. By definition of the semantics, there exists an element $e \in D_I$ s.t. $[\gamma_i] = true$ where $I' = I\{x \leftarrow e\}$. We apply the induction hypothesis on the formulae γ, γ' and on the interpretation I' to deduce that $[\gamma']_{I'} = true$ (note that we have $I' \models \psi \Rightarrow \psi'$, since I' and I are identical on non variable symbols). Therefore, $[\phi']_I = true$.

  $\forall$: The proof is similar if ϕ is of the form $\forall x \gamma$.

■

[1] not only in ϕ, but also in the whole formula containing ϕ

Lemma 15 *The rule $CF_\exists$ preserves satisfiability, i.e. if ψ, ψ' are two formulae and if ψ' is obtained from ψ by applying the rule $CF_\exists$ then ψ is satisfiable iff ψ' is satisfiable. Moreover, any model of ψ' is a model of ψ.*

Proof

By definition ψ' is obtained from ψ by replacing a formula $\exists x\phi$ by another formula $\phi\{x \rightarrow f(y_1, \ldots, y_n)\}$, where $y_1, \ldots, y_n$ are the free variables in $\exists x\phi$ and f is a symbol not occurring in ψ.

- Assume that ψ' is satisfiable. Then there exists an interpretation I s.t. $I \models \psi'$. By Corollary 11, the formula $\phi\{x \rightarrow f(y_1, \ldots, y_n\} \Rightarrow \exists x\phi$ is true in any interpretation, in particular in I. Thus we can apply Lemma 14 and we deduce that $I \models \psi$.

- Assume that ψ is satisfiable. Then there exists an interpretation I s.t. $I \models \psi$. We now construct an interpretation J, satisfying ψ'. J is identical to I, except for the interpretation of f. In order to define the interpretation of f, we have to specify the value of $f_J(e_1, \ldots, e_n)$, for each n-tuple $(e_1, \ldots, e_n) \in D_I^n$. We choose for $f_J(e_1, \ldots, e_n)$ an element e (arbitrarily chosen) s.t. $I[x \leftarrow e, y_1 \leftarrow e_1, \ldots, y_n \leftarrow e_n] \models \phi$. If no such element exists, then the interpretation of $f_J(e_1, \ldots, e_n)$ is chosen arbitrarily.

  By construction, $J \models \exists x\phi \Rightarrow \phi\{x \rightarrow f(y_1, \ldots, y_n)\}$. Moreover $J \models \phi$ (since $I \models \phi$, f does not occur in ϕ and I, J are identical on any symbol distinct from f). Thus we can apply Lemma 14 and we deduce that $J \models \psi'$.

■

We denote by CF the set of rules: $CF_\Rightarrow$, $CF_\Leftrightarrow$, $CF_{\neg\neg}$, $CF_{\neg\wedge}$, $CF_{\neg\vee}$, $CF_{\neg\exists}$, $CF_{\neg\forall}$, $CF_{\vee\wedge}$, $CF_{\vee\forall}$, $CF_{\forall\wedge}$ and $CF_\exists$

Lemma 16 *The non deterministic application of the rules in CF terminates.*

Proof We introduce a measure m_{CF}, mapping any first-order formula ϕ to a natural number $m_{CF}(\phi)$ and s.t. the value of m_{CF} decreases each time a rule in CF is applicable. Since the measure is positive, it cannot decrease indefinitely hence there is no infinite derivation.

$$
\begin{array}{lcll}
m_{CF}(\phi) & \stackrel{\text{def}}{=} & 2 & \text{If } \phi \text{ is atomic} \\
m_{CF}(\phi \Leftrightarrow \psi) & \stackrel{\text{def}}{=} & 1 + m_{CF}((\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)) & \\
m_{CF}(\phi \Rightarrow \psi) & \stackrel{\text{def}}{=} & 1 + m_{CF}(\neg\phi \vee \psi) & \\
m_{CF}(\phi \vee \psi) & \stackrel{\text{def}}{=} & m_{CF}(\phi) \times m_{CF}(\psi) & \\
m_{CF}(\phi \wedge \psi) & \stackrel{\text{def}}{=} & 1 + m_{CF}(\phi) + m_{CF}(\psi) & \\
m_{CF}(\neg\phi) & \stackrel{\text{def}}{=} & m_{CF}(\phi)^{m_{CF}(\phi)} & \\
m_{CF}(\exists x\phi) & \stackrel{\text{def}}{=} & 1 + m_{CF}(\phi) & \\
m_{CF}(\forall x\phi) & \stackrel{\text{def}}{=} & 2 \times m_{CF}(\phi) + 1 &
\end{array}
$$

The fact that m really decreases is left as an exercise to the reader. It suffices to check, for each rule in CF, that the value of the left-hand side is strictly

greater than the value of the right-hand side. Notice that by construction, we have $m(\phi) \geq 2$ for every formula ϕ. ■

Lemma 17 *Any closed formula that is irreducible by the rules in CF is a conjunction of (universally quantified) clauses.*

Proof If a negation occurs in the front of a complex subformula then one of the negation rules applies. Thus the negations only occur in literals.

By irreducibility w.r.t. the rules $CF_{\Rightarrow}, CF_{\Leftrightarrow}$ and $CF_{\exists}$, the formula only contains the connectives $\neg, \vee, \wedge, \forall$ (otherwise one of the above rules necessarily applies, which is impossible since the formula is assumed to be irreducible).

If a conjunction or a universal quantifier occurs in a disjunction then the rule $CF_{\vee\wedge}$ or $CF_{\vee\forall}$ applies. Thus the only disjunctive subformulae must be disjunctions of literals.

Finally, if a universal quantifier occurs behind a conjunction then the rule $CF_{\forall\wedge}$ applies.

Hence the formula is a conjunction (possibly empty, and possibly of length 1) of (universally quantified) clauses.

In order to obtain a set of clauses, it suffices to replace the conjunction of formulae by a set of formulae and to remove all universal quantifiers (this obviously preserves equivalence since free variables are implicitly universally quantified).

■

2.3 Complexity and Renaming

The clausal transformation algorithm can increase the size of the formula, as evidenced by the following example:

$$\phi = \bigvee_{i=1}^{n} (a_i \wedge b_i)$$

Applying the Distributivity on ϕ produces the following set of clauses:

$$\begin{array}{l} a_1 \vee a_2 \vee \ldots \vee a_{n-1} \vee a_n \\ a_1 \vee a_2 \vee \ldots \vee a_{n-1} \vee b_n \\ a_1 \vee a_2 \vee \ldots \vee b_{n-1} \vee a_n \\ a_1 \vee a_2 \vee \ldots \vee b_{n-1} \vee b_n \\ \ldots \\ b_1 \vee b_2 \vee \ldots \vee b_{n-1} \vee a_n \\ b_1 \vee b_2 \vee \ldots \vee b_{n-1} \vee b_n \end{array}$$

2^n clauses are produced. Thus the transformation algorithm is at least exponential. A similar explosion of the size of the formula may happen with equivalences: the reader should try for instance to compute the clausal form of the formula $(a_1 \Leftrightarrow (a_2 \Leftrightarrow \ldots (a_{n-1} \Leftrightarrow a_n)))$.

In order to reduce the complexity, it is necessary to avoid the duplication of subformulae. This can be done by introducing additional predicate symbols, in order to give a "name" to a given subformula.

More precisely, a formula ϕ of free variables $x_1, \ldots, x_n$ can be replaced by an atom $P(x_1, \ldots, x_n)$, where the equivalence $P(x_1, \ldots, x_n) \Leftrightarrow \phi$ is added into the formula as an axiom. The interest is that $P(x_1, \ldots, x_n)$ may be reused several times in the formula, without having to repeat the whole formula ϕ.

Formally, this is done by the following rule:

$$\psi \quad \rightarrow \quad \psi' \wedge \forall x_1, \ldots, x_n (P(x_1, \ldots, x_n) \Leftrightarrow \phi)$$

where ϕ is a subformula of ψ, $x_1, \ldots, x_n$ is the set of free variables in ϕ and ψ' is obtained from ψ by replacing any subformula of the form $\psi\{x_i \rightarrow t_i \mid i \in [1..n]\}$ by $P(t_1, \ldots, t_n)$ (in particular, if $n = 0$ then P is a propositional variable).

By applying this rule before the distributivity rule or equivalence rule in order to avoid duplicating complex formulae it is possible to obtain a transformation algorithm which is linear w.r.t. the size of the initial formula (in number of clauses, quadratic in size).

For instance, the new distributivity rule may be written as follows:

$$\phi \vee (\psi_1 \wedge \psi_2) \quad \rightarrow \quad (P(x_1, \ldots, x_n) \vee \psi_1) \wedge (P(x_1, \ldots, x_n) \vee \psi_2)$$

If ϕ is a complex formula of free variables $x_1, \ldots, x_n$ and P is a new predicate symbol. The axiom $\forall x_1, \ldots, x_n (P(x_1, \ldots, x_n) \Leftrightarrow \phi)$ is added to the whole formula (of course it must also be transformed into clausal form).

For instance, applying the previous naming rule on the formula $\phi = \bigvee_{i=1}^{n} (a_i \wedge b_i)$ produces the following clause form:

$$\bigvee_{i=1}^{n} p_i \wedge \bigwedge_{i=1}^{n} [(\neg p_i \vee a_i) \wedge (\neg p_i \vee b_i) \wedge (\neg a_i \vee \neg b_i \vee p_i)].$$

p_i is a "name" for the formula $a_i \wedge b_i$. $(\neg p_i \vee a_i) \wedge (\neg p_i \vee b_i) \wedge (\neg a_i \vee \neg b_i \vee p_i)$ is the clausal form of $p_i \Leftrightarrow a_i \wedge b_i$.

Systematic application of the renaming rule on each subformula is called a *structural* clausal transformation [10]. The reader can consult [3] or [9] for more details.

3 Unification

The goal of the unification algorithm is, given two terms t and s, to check whether there exists a substitution σ s.t. the terms $t\sigma$ and $s\sigma$ are syntactically equal (and if it is the case to compute the corresponding set of substitutions σ). If $t\sigma = s\sigma$ then σ is called a *unifier* of t and s, and two terms having a unifier are said to be *unifiable*.

For instance, $f(x,a)$ and $f(b,y)$ are unifiable and the (unique) unifier is $\{x \to b, y \to a\}$ (a, b denote distinct constant symbols and x, y are variables). The terms $f(x,a)$ and $f(b,x)$ have no unifier (since x cannot be equal to a and b simultaneously). Similarly, x and $f(x)$ have no unifier (since a term cannot strictly occur in itself) and $f(x)$ and $g(y)$ are unifiable only if $f = g$. $f(x,y)$ and $f(u,v)$ have several unifiers: $\{x \mapsto u, y \mapsto v\}$ is a unifier, but also $\{x \mapsto a, y \mapsto a, u \mapsto a, v \mapsto a\}$, ... (actually there exist an infinite number of distinct unifiers).

More formally, a *unification problem* is a formula of the form $\bigwedge_{i=1}^{n} \phi_i$ (with possibly $n = 1$), where for every $i \in [1..n]$ ϕ_i is either *false* or *true* or an equation of the form $t_i \doteq s_i$. A substitution σ is said to be a *solution* of $\bigwedge_{i=1}^{n} \phi_i$ if for every $i \in [1..n]$ either $\phi_i = true$ or $\phi_i = (t_i \doteq s_i)$ and $t_i\sigma = s_i\sigma$. The set of solutions of a problem ϕ is denoted by $sol(\phi)$.

We denote by R_{unif} the following set of rules, operating on unification problems:

Trivial	$t \doteq t$	$\to$	*true*
Occur Check	$x \doteq t$	$\to$	*false*
	If $x \in Var(t)$ and $x \neq t$		
Decomposition	$f(t_1, \ldots, t_n) \doteq f(s_1, \ldots, s_n)$	$\to$	$\bigwedge_{i=1}^{n} t_i \doteq s_i$
Clash	$f(t_1, \ldots, t_n) \doteq g(s_1, \ldots, s_m)$	$\to$	*false*
	If $f \neq g$		
Replacement	$x \doteq t \wedge \phi$	$\to$	$x \doteq t \wedge \phi\{x \to t\}$
	If x is a variable not occurring in t, x occurs in ϕ		
	and either t is not a variable or t occurs in ϕ		
Simplification	$true \wedge \phi$	$\to$	ϕ
Failure	$false \wedge \phi$	$\to$	*false*

The rules are to be applied modulo the commutativity of $\doteq$ and the commutativity and associativity of $\wedge$. For instance, the problem $z \doteq g(x) \wedge a \doteq x \wedge y \doteq f(x)$ may be reduced to $z \doteq g(a) \wedge a \doteq x \wedge y \wedge f(a)$.

Lemma 18 *The non deterministic application of the rules in R_{unif} terminates on every unification problem ϕ.*

Proof A variable x is said to be *solved* in a unification problem ψ if ψ is of the form $x \doteq t \wedge \psi'$, where x occurs neither in t nor in ψ'.

We introduce the following measure m_{unif} on unification problems. $m_{unif}(\phi) = (v, s)$ where v is the number of unsolved variables in ϕ and s is the size of ϕ (i.e. the number of symbols of $\mathcal{L}$ occurring in ϕ). The reader can easily check that no rule can increase v. Moreover, all the rules, except the Replacement rule, strictly decrease s. The Replacement rule may increase s (since t is duplicated), but decreases v strictly (since x becomes solved).

Thus m_{unif} decreases each time a rule in R_{unif} is applied. Since m_{unif} is bounded it cannot decrease indefinitely, hence R_{unif} terminates. ■

Lemma 19 *Let ϕ be a unification problem and let ψ be a problem obtained by applying a rule in R_{unif} on ϕ. $sol(\phi) = sol(\psi)$.*

Proof It is easy to check that each rule preserves the set of solutions. The detailed proof is left as an exercise for the reader. ■

A unification problem ϕ is said to be *solved* if it is either *false* or *true* or a formula of the form $\bigwedge_{i=1}^{n} x_i \doteq t_i$ where the variables $x_1, \ldots, x_n$ occurs only once in ϕ.

If ϕ is *false* then it has no solution. If ϕ is *true* then any substitution is a solution. If ϕ is $\bigwedge_{i=1}^{n} x_i \doteq t_i$, then it is clear that ϕ has an obvious solution: the substitution $\{x_i \rightarrow t_i \mid i \in [1..n]\}$. Moreover, this substitution is also a most general one, i.e. any solution of ϕ is an instance of the previous substitution.

Lemma 20 *If ϕ is irreducible by the rules in R_{unif} then ϕ is solved.*

Proof ϕ is of the form $\bigwedge_{i=1}^{n} \phi_i$. By irreducibility w.r.t. the simplification and failure rules, if $\phi_i = \mathit{false}$ or $\phi_i = \mathit{true}$ for some $i \in [1..n]$ then we must have $i = 1$ and in this case we have $\phi = \mathit{false}$ or $\phi = \mathit{true}$ thus ϕ is solved.

Thus we assume that for every $i \in [1..n]$ ϕ_i is of the form $t_i \doteq s_i$. If the head symbol of both t_i and s_i are function symbols, then either the Decomposition rule or the Clash rule applies (if the head symbols are the same the the Decomposition rule applies, otherwise the Clash rule applies). Consequently, one of the term t_i or s_i must be a variable. We assume, w.l.o.g. that t_i is a variable x_i. If x_i occurs in s_i then either the rule Trivial applies (if $x_i = t_i$) or the Occur Check applies (if $x_i \neq t_i$). Thus x_i does not occur in t_i. Finally, if x_i occurs elsewhere in the formula, then the Replacement rule applies.

Therefore ϕ is solved. ■

The previous results show that every unification problem having a solution has a most general solution (i.e. a solution that is more general than any solution). In particular, if t, s are two unifiable terms, then t and s have a most general unifier (m.g.u.) σ, i.e. a substitution s.t. any unifier of t and s is an instance of σ. Clearly, this m.g.u. is unique, up to a renaming of variables. The rules in R_{unif} provide an algorithm to check whether two terms are unifiable or not and if possible to compute the m.g.u..

Complexity

The unification algorithm is exponential, as evidenced by the following example: $t = f(x_1, x_2, \ldots, x_n)$, $s = f(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}))$. We obtain the substitution: $x_1 \rightarrow f(x_0, x_0)$, $x_2 \rightarrow f(f(x_0, x_0), f(x_0, x_0))$, $x_3 \rightarrow f(f(f(x_0, x_0), f(x_0, x_0)), f(f(x_0, x_0), f(x_0, x_0)))$ etc.

The size of the obtained term is exponential in n.

However, this high complexity can be easily reduced by using structure sharing: identical terms can be *shared* between the terms in which they occur instead to being duplicated. This is done by using directly acyclic graphs (DAG) in order to represent complex terms, which can easily be encoded by pointers or references.

Using this convention, it is easy to show that the unification algorithm is polynomial (see [1] for more details). Informally, the number of *distinct* subterms does not increase.

4 Herbrand Theorem

A term or a clause is said to be *ground* if it contains no variables. A substitution σ is said to be ground if for every $x \in dom(\sigma)$, $x\sigma$ is ground.

The Herbrand Theorem (by the French Mathematician Jacques Herbrand, 1908-1931) relates the satisfiability of a set of clauses S to the one of the set of its ground instances.

If S is a set of clauses then S_{inst} denotes the set of *ground instances* of S, i.e. the set of clauses $C\sigma$, s.t. C is a clause in S, and σ is a ground substitution of domain $Var(C)$.

We assume that the language $\mathcal{L}$ contains at least a constant symbol. In this case, S_{inst} is necessarily non empty, if $S \neq \emptyset$.

Definition 21 *(Herbrand Interpretation) An interpretation I is said to be an Herbrand interpretation iff:*

- *Its domain is the set of ground terms (it is non empty if the the language contains at least a constant symbol).*
- *For every function symbol f of arity n and for every n-tuple $(t_1, \ldots, t_n)$ of ground terms, we have $f_I(t_1, \ldots, t_n) \stackrel{def}{=} f(t_1, \ldots, t_n)$.*

By the previous definition, the domain and the interpretation of function symbols are fixed. Thus a Herbrand interpretation I is uniquely defined by giving the interpretation of predicate symbols, i.e. by specifying the set of ground atoms that are true in I. Thus a Herbrand interpretation is often seen as a set of ground atoms.

Moreover the interpretation of a ground term t in a Herbrand interpretation is t itself:

Lemma 22 *For any ground term t and for any Herbrand interpretation I, $[t]_I \stackrel{def}{=} t$.*

Proof By a straightforward induction on the size of t. ■

This implies that a clause is false in a Herbrand interpretation iff it has a ground instance that is false:

Lemma 23 *If C is a clause and I is a Herbrand interpretation s.t. $I \not\models C$, then there exists a ground substitution σ of domain $Var(C)$ s.t. $I \not\models C\sigma$.*

Proof

Let $x_1, \ldots, x_n$ be the free variables in C. We have $I \not\models C$, thus by definition there exist $v_1, \ldots, v_n \in D_I$ s.t. $[C]_J = false$, where $J = I\{x_i \leftarrow v_i \mid i \in [1..n]\}$. By definition, $v_1, \ldots, v_n$ are ground terms. Moreover by Lemma 22, $[v_i]_J = v_i$.

Let $\sigma = \{x_i \mapsto v_i \mid i \in [1..n]\}$. By Lemma 10, we have $[C]_J = [C\sigma]_J$. Since $x_1, \ldots, x_n$ do not occur in $C\sigma$ we have $[C\sigma]_J = [C\sigma]_I$. Thus $[C\sigma]_I = \mathit{false}$, hence $I \not\models C\sigma$. ■

Theorem 24 *(Herbrand) Let S be a set of clauses. S is unsatisfiable iff S_{inst} is unsatisfiable. Moreover if S is satisfiable then S has a Herbrand model.*

Proof Clearly, every instance of a clause occurring in S is a logical consequence of S. Thus $S \models S_{inst}$ and if S is satisfiable then S_{inst} is also satisfiable.

Now, assume that S_{inst} is satisfiable. Then S_{inst} has a model I. Let J be the Herbrand interpretation s.t. for any n-ary predicate symbol P and for any n-tuple of ground terms $t_1, \ldots, t_n$, $P_J(t_1, \ldots, t_n) = \mathit{true}$ iff $I \models P(t_1, \ldots, t_n)$. By Lemma 22, this implies that $J \models P(t_1, \ldots, t_n)$ iff $I \models P(t_1, \ldots, t_n)$. Thus for every ground literal L, we have $J \models L$ iff $I \models L$.

Assume that $J \not\models S$. Then there exists a clause $C \in S$ s.t. $J \not\models C$. By Lemma 23 there exists a ground substitution σ of the variables in C s.t. $J \not\models C\sigma$. But $C\sigma \in S_{inst}$ thus $I \models C\sigma$. There exists a literal L in C s.t. $I \models L\sigma$. But then by definition of J we have $J \models L\sigma$ thus $J \models C\sigma$, which is impossible.

Thus $J \models S$. ■

A clause is said to be *unit* if it contains exactly one literal. The next theorem introduces an efficient way of checking the satisfiability of a set of unit clauses. The class of sets of unit clauses is called the *Herbrand class*.

Theorem 25 *A set of unit (pairwise variable-disjoint) clauses S is unsatisfiable iff it contains two unit clauses L and $\neg L'$, s.t. L, L' are unifiable.*

Proof If S is unsatisfiable then S_{inst} is unsatisfiable (by Herbrand's theorem).

Assume that S_{inst} contains two complementary unit clauses A and $\neg A$. Then by definition S contains two clauses L and $\neg L'$ s.t. there exist two ground substitutions σ and θ of domains $\mathit{Var}(L)$ and $\mathit{Var}(L')$ respectively s.t. $L\sigma = A$ and $L'\theta = A$. Since $L, \neg L'$ are variable-disjoint, we can define the substitution $\eta = \sigma \cup \theta$. η is a unifier of L, L', thus L, L' are unifiable.

Now assume that S_{inst} does not contain any pair of complementary literals. We consider the Herbrand interpretation I s.t. $I \models A$ iff $A \in S_{inst}$. Obviously, I validates all positive literals in S_{inst} (by definition). Moreover, if $\neg A$ is a negative literal in S_{inst} s.t. $I \not\models \neg A$, then we have $I \models A$, thus $A \in S_{inst}$, which contradicts our assumption since S_{inst} would contain A and $\neg A$. Thus $I \models S_{inst}$ which contradicts the fact that S_{inst} is unsatisfiable.

■

5 The Resolution Calculus

The Resolution calculus (initially introduced by Robinson [13]) is a proof procedure for first-order formulae in clausal form, especially developed for automated

theorem proving. It is the most efficient and commonly used proof procedure (for non equational theorem proving) and it is the basis of several efficient theorem provers. In contrast to other logical calculi such as the sequent calculus or the semantic tableaux, it is very uniform (only two inference rules) which makes the implementation and control much easier.

The Resolution calculus provides a (non terminating) algorithm for checking whether a given set of clauses S is unsatisfiable. It works by **refutation** (*reductio at absurdum*): **inference rules** are repeatedly applied on S in order to derive new clauses (that are all logical consequences of S), until a contradiction (i.e. an empty clause $\Box$) is obtained (which immediately entails that S is unsatisfiable). **Selection strategies** (for reducing the branching of the procedure) and **redundancy elimination rules** (for eliminating useless clauses) help to reduce the search space.

This algorithm is only a **semi-decision procedure**: if the set of clauses is unsatisfiable then a contradiction is eventually obtained, but if S is satisfiable then an infinite number of clauses may be deduced and the algorithm may run forever. As we shall see in the last part of the course, there is no decision procedure for first-order logic.

5.1 Inference Rules

The Resolution calculus is defined by the two following inference rules: Resolution and Factorisation (they are sometimes combined into a unique rule).

We assume given a **selection function** *sel*, mapping each clause C to a set of literals occurring in C. These literals are said to be *selected*. We will show the usefulness of this selection function latter. Intuitively selected literals are the only literals on which one has the right to apply the inference rules. Thus *sel* is used to control (and to restrict) the application of the inference rules. In the most simple case, $sel(C)$ may be defined as the set of literals occurring in C (= no control).

5.1.1 Resolution

A clause R is said to be a *sel-resolvent* of two clauses C and D, if the following conditions hold:

- C and D share no variable (i.e. $Var(C) \cap Var(D) = \emptyset$).
- C and D are of the form $L \vee C'$ and $\neg L' \vee D'$ respectively. C' and D' may be empty, then we have $C = L$ or $D = \neg L'$.
- L and L' are unifiable, with a m.g.u. σ.
- $L\sigma$ and $\neg L'\sigma$ are selected in $C\sigma$ and $D\sigma$ respectively.
- R is $(C' \vee D')\sigma$.

Notice that Condition 1 is essential. If the clauses share variables, then one has to rename them before applying the Resolution rule.

Alternatively, the rule may be depicted as follows (with the same meaning):

$$\frac{L \vee C \qquad \neg L' \vee D}{(C \vee D)\sigma}$$

where σ is the m.g.u. of L and L' and where $L\sigma \in sel((L \vee C\sigma))$, $\neg L'\sigma \in sel((\neg L' \vee D)\sigma)$.

5.1.2 Factorisation

A clause F is said to be a *sel-factor* of a clause C iff the following holds:

- C is of the form $L \vee L' \vee D$.
- L and L' are unifiable, with a m.g.u. σ.
- $L\sigma$ is selected in $C\sigma$.
- $F = (L \vee D)\sigma$.

Notice that L, L' may be negative. Alternatively, the rule may be depicted as follows:

$$\frac{L \vee L' \vee C}{(L \vee C)\sigma}$$

where σ is the m.g.u. of L and L' and where $L\sigma \in sel((L \vee L' \vee C)\sigma)$.

5.1.3 Immediate Consequence Operator

As usual, the AC properties of $\vee$ must be taken into account. For instance the resolution rule is applicable between $P \vee Q$ and $R \vee \neg Q$, and deduces $P \vee R$.

A set of clauses S is said to be *variable-disjoint* iff for every pair $(C, D) \in S^2$, if $C \neq D$ then $Var(C) \cap Var(D) = \emptyset$ (i.e. the clauses in S share no variable).

Note that for every set of clauses S, there exists a set of clauses S' that is equivalent to S and variable-disjoint: it suffices to rename the variables that are shared by distinct clauses.

If S is a variable-disjoint set of clauses, we denote by $Res_{sel}(S)$ the set of *sel*-resolvent of clauses in S. If S is not variable-disjoint then $Res_{sel}(S)$ is defined as the set $Res_{sel}(S')$ where S' is a variable-disjoint set of clauses equivalent to S (the renaming is chosen arbitrarily).

We denote by $Fact_{sel}(S)$ the set of *sel*-factors of clauses in S and by $D_{sel}(S)$ the set $D_{sel}(S) \stackrel{\text{def}}{=} Res_{sel}(S) \cup Fact_{sel}(S)$.

Example 26 *Assume for instance that* $S = \{P(x, a) \vee Q(x), \neg P(b, y) \vee R(y), R(u) \vee R(v) \vee R'(u, v)\}$ *and that* $sel(C) = C$ *for any clause* C. *S is variable-disjoint. The reader can check that* $D_{sel}(S)$ *is* $\{Q(b) \vee R(a), R(u) \vee R'(u, u)\}$.

*If $S' = \{P(x) \vee Q(y), \neg P(x) \vee Q(y)\}$ then S' is not variable-disjoint. We have $D_{sel}(S') = \{Q(y) \vee Q(y')\}$ (and **not** $\{Q(y) \vee Q(y)\}$!).*

Since the two clauses in S' share variables, one has to rename them before applying the Resolution rule. We obtain the set $\{P(x) \vee Q(y), P(x') \vee Q(y')\}$, on which the Resolution rule can be applied. Of course there are many ways of performing the renaming, but the obtained clauses are all equivalent.

A *derivation* from S is a sequence of clauses $C_1, \ldots, C_n$ s.t. for any $i \in [1..n]$, $C_i \in D_{sel}(S \cup \{C_1, \ldots, C_{i-1}\})$. We write $S \vdash_{sel} C$ if there exists a derivation $C_1, \ldots, C_n$ from S s.t. $C_n = C$.

5.2 Soundness

The following theorem shows that the rules are correct (or *sound*), i.e. that all the clauses that are deduced using the previous rules are logical consequences of the premises.

Theorem 27 *Let S be a set of clauses. $S \models D_{sel}(S)$.*

Proof Let $C \in D_{sel}(S)$. Let I be a model of S. We have to show that $[C]_I = true$.

C is either a resolvent of two clauses in S or a factor of a clause in S. We consider the two cases separately.

- If C is deduced by resolution, then C is of the form $(C' \vee D')\sigma$, where S contains two clauses of the form $L \vee C'$ and $\neg L' \vee D'$, and where $L\sigma = L'\sigma$. We have $I \models S$ hence $I \models L \vee C'$ and $I \models \neg L' \vee D'$. By Lemma 11, $I \models (L \vee C')\sigma$ and $I \models (\neg L' \vee D')\sigma$. Thus $[(L \vee C')\sigma]_I = [(\neg L' \vee C')\sigma]_I = true$. If $[L\sigma]_I = true$ then we have $[\neg L'\sigma]_I = false$, hence $[D'\sigma]_I = true$. Otherwise, we have $[C'\sigma]_I = true$. In both cases we have $[C'\sigma \vee D'\sigma]_I = true$, i.e. $[C]_I = true$.

- If C is deduced by factorisation: the proof is left as an exercise to the reader.

■

Corollary 28 *If $S \vdash_{sel} \Box$ then S is unsatisfiable.*

Proof Using Theorem 27, we can easily show that for any derivation $C_1, \ldots, C_n$ from S, we have $S \models C_n$ (by an easy induction on n). Moreover, if $S \models \Box$ then clearly S is unsatisfiable, since $\Box$ is equivalent to *false*. ■

As we shall see the converse of Corollary 28 also holds: if S is unsatisfiable then $S \vdash_{sel} \Box$. This property is called *refutational completeness*. It ensures that if a set of clauses in unsatisfiable, then a contradiction will eventually be found by applying the Resolution and Factorisation rule.

5.3 Redundancy Elimination Rules

In this section, we introduce some criteria for detecting and removing useless clauses (this is called *redundancy elimination*).

Definition 29 *A clause is said to be a* tautology *if it is of the form* $L \vee \neg L \vee C$.

Clearly, all tautology are valid (and all valid clauses are tautologies).

Definition 30 *A clause* C *is said to be* subsumed *by a clause* D *if there exists a substitution* σ *s.t.* C *is of the form* $D\sigma \vee D'$ *(modulo the AC properties of* $\vee$*)*[2].

Obviously, this implies that $D \models C$ (but the converse does not hold: for instance $P(x) \vee \neg P(f(f(x)))$ is a logical consequence of $P(x) \vee \neg P(f(x))$ but is not subsumed).

Definition 31 *A clause* C *is said to be* redundant *w.r.t. a set of clauses* S *iff* C *is either a tautology or subsumed by a clause in* S*. It is said to be* strictly redundant *w.r.t. a set of clauses* S *iff either* C *is a tautology or if it is subsumed by a clause* D *in* S *s.t.* C *does not subsume* D*.*

As we shall see, clauses that are strictly redundant are useless, in the sense that they are not needed for deriving the empty clause. Thus they can be simply ignored.

6 Refutational Completeness of the Resolution Calculus

A set of clauses S is said to be *sel-saturated* (or simply saturated, if *sel* is implicit) iff every clause $C \in D_{sel}(S)$ is redundant w.r.t. S.

In this section we shall show that any saturated clause set not containing $\Box$ is satisfiable. This implies that the Resolution calculus is refutationally complete: if one computes the whole set of clauses that can be derived from a given set of clauses S by Resolution and Factorisation (i.e. the set $\{C \mid S \vdash_{sel} C\}$, then this set is obviously saturated, thus it is unsatisfiable iff it contains $\Box$. Some additional conditions are required on the selection function in order to ensure the desired property.

The redundancy criteria shows that the clauses that are redundant w.r.t. clauses that have already been deduced are not useful (hence can be ignored).

6.1 Ground Case

In this section we handle the particular case in which the clauses are ground (i.e. contain no variable).

[2] If C, D are viewed as sets then we may write $D\sigma \subseteq C$.

We assume that a total ordering on ground atoms $<$ is given. This ordering is extended to literals simply by ignoring the negation symbol: $\neg A < B$ iff $A < B$ iff $A < \neg B$.

A literal L is said to be *maximal* in a clause C if there is no literal L' in C s.t. $L' > L$.

We assume that the selection function satisfies the following property: **for any clause C, either $sel(C)$ contains all maximal literals in C, or $sel(C)$ contains at least a negative literal in C.**

Lemma 32 *Let S be a set of ground clauses. If S is saturated and unsatisfiable then S contains $\Box$.*

Proof Assume that S is saturated and does not contain $\Box$. We construct an Herbrand interpretation I that satisfies S. I is constructed by specifying the value of each atom in I. This is done by induction on the ordering $<$.

Let A be an atom. Assume that the value of any atom $B < A$ has been constructed. We distinguish two cases:

- If there exists a clause $L_1 \vee \ldots \vee L_n \vee A \in S$ s.t. A is selected in $L_1 \vee \ldots \vee L_n \vee A$ and for every $i \in [1..n]$ we have $A > L_i$ and $[L_i]_I = \mathit{false}$, then $[A]_I \stackrel{\text{def}}{=} \mathit{true}$. Notice that $L_1, \ldots, L_n$ have a truth value at this point since they are strictly smaller than A.
- Otherwise, $[A]_I = \mathit{false}$.

We show that $I \models S$. Let C be a clause in S. Assume that C is not true in I. Let C be the smallest clause having this property (w.r.t. to the ordering $<$, i.e. we choose, among all the clauses of S that are false in I, the one containing the smallest possible literals). This means that any clause $D \in S$ s.t. D is smaller than C is true in I.

We claim that for any clause D that is smaller than C and redundant w.r.t. S, we have $I \models D$. Indeed, either D is a tautology (then D is valid and $I \models D$) or S contains a clause D' subsuming D. But in this case D' must be also smaller than C (since D' is a subclause of D and D is smaller than C) thus we have $I \models D'$. Since $D' \models D$, we deduce $I \models D$.

Now, assume that $sel(C)$ contains a negative literal $\neg L$. Then $C = \neg L \vee C'$. Since $I \not\models C$, we have $I \not\models C'$ and $I \models L$. By definition of I, $I \models L$ implies that S contains a clause of the form $L_1 \vee \ldots \vee L_n \vee L$ s.t. L is selected, and for every $i \in [1..n]$ we have $I \models \neg L_i$ and $L_i < L$. Then the Resolution rule applies on this clause and C and generates the clause: $L_1 \vee \ldots \vee L_n \vee C'$. Since S is saturated, this clause must be redundant in S. We have $I \not\models L_1 \vee \ldots \vee L_n \vee C'$ (since $I \models \neg L_i$ and $I \not\models C'$). Moreover, $L_1 \vee \ldots \vee L_n \vee C'$ is strictly smaller than C (since a literal $\neg L$ has been replaced by n literals $L_1, \ldots, L_n$ that are strictly smaller). By definition of C this is impossible.

Thus $sel(C)$ contains no negative literal. Consequently, $sel(C)$ contains all the maximal literals in C. Let L be a maximal literal in C (L exists since C is non empty). Obviously, L is positive. We have $C = L \vee C'$.

If C' contains a literal that is not smaller than L, then either C' contains L or C' contains $\neg L$ (since $<$ is total on ground atoms). If C' contains $\neg L$ then C is valid, which is impossible since $I \not\models C$. If C' contains L, it is of the form $C' = L \vee C''$ then the Factorisation rule applies on C and generates a clause $L \vee C''$. Since S is saturated, $L \vee C''$ is redundant in S, which is impossible since $I \not\models L \vee C''$ and $L \vee C''$ is strictly smaller than C (an occurrence of a literal L has been deleted).

Consequently, every literal L' in C' is strictly smaller than L. Moreover, we have $I \not\models L'$ (since $I \not\models C$) and L is selected in C. By definition of I, this implies that $[L]_I = \mathit{true}$, thus $I \models C$, which contradicts the definition of C. ■

6.2 Non Ground Case

We now extend the previous result to the case in which the clauses contain variables. For the non ground case, we assume that the selection function satisfies the following property (Lifting property): **if $L \vee C$ is a clause and if there exists a substitution η s.t. $L\eta$ is selected in $(L \vee C)\eta$, then L must be selected in $L \vee C$.**

This property entails that the inferences that can be performed at the ground level (i.e. on ground instances of the considered set of clauses) can be "lifted" to the non ground level. More precisely, the clauses that can be obtained by applying the inference rules on ground instances of a set of clauses S are instances of clauses that can be deduced from S. More formally:

Lemma 33 *Let S be a set of clauses. For every clause $C \in D_{sel}(S_{inst})$, there exists a clause $D \in D_{sel}(S)$ s.t. C is an instance of D.*

Proof C is obtained by applying either the Resolution rule or the Factorisation rule. We distinguish two cases:

- If C is obtained by applying the Resolution rule, then C is of the form $C' \vee D'$, where S_{inst} contains two clauses $L \vee C'$ and $\neg L \vee D'$ (since the clauses are ground there is no unifier). Moreover, L and $\neg L$ are selected in $L \vee C'$ and $\neg L \vee D'$ respectively. By definition, since $L \vee C'$ and $\neg L \vee D'$ occurs in S_{inst}, they must be instances of some clauses in S. Thus there exist two clauses $L' \vee C''$ and $\neg L'' \vee D''$ in S and two substitutions θ and θ' s.t. $L'\theta = L, L''\theta' = L, C''\theta = C'$ and $D''\theta' = D'$. We assume, w.l.o.g, that $L' \vee C''$ and $\neg L'' \vee D''$ share no variable (this is possible since the shared variables are renamed before applying the Resolution rule). Then $\eta = \theta \cup \theta'$ is a unifier of L' and L''. Let σ be the m.g.u. of L', L''. By definition, η is an instance of σ, i.e. there exists a substitution η' s.t. $\eta = \sigma\eta'$.

  By the Lifting property on *sel*, since L is selected in $L \vee C'$ and since $(L' \vee C'')\sigma\eta' = (L \vee C')$, $L'\sigma$ must be selected in $(L' \vee C'')\sigma$. Similarly, $\neg L''\sigma$ is selected in $(\neg L'' \vee D'')\sigma$. Consequently, the Resolution rule is applicable between $L' \vee C''$ and $\neg L'' \vee D''$. The resolvent is $(C'' \vee D'')\sigma$. Thus $(C'' \vee D'')\sigma \in D_{sel}(S)$. Moreover, $(C'' \vee D'')\sigma\eta' = (C \vee D)$.

- If C is obtained by the Factorisation rule: the proof is similar (it is left to the reader).

■

This property is essential for completeness. It implies the following:

Lemma 34 *If a set of clauses S is saturated, then S_{inst} is saturated.*

Proof Let C be a clause in $D_{sel}(S_{inst})$. By Lemma 33, there exist a clause $D \in D_{sel}(S)$ and a substitution σ s.t. $D\sigma = C$. By definition of the notion of saturated set, D is redundant in S. If D is a tautology then $D\sigma = C$ is also a tautology, thus C is redundant in S_{inst}. Otherwise, there exists a clause D' in S that subsumes D, i.e. there exists a substitution θ s.t. $D = D'\theta \vee D''$, for some clause D''. Then $C = D\sigma = D'\theta\sigma \vee D''\sigma$. By definition $D'\theta\sigma \vee D''\sigma$ is in S_{inst} thus C is subsumed by a clause in S_{inst} hence is redundant. ■

We deduce easily the following:

Theorem 35 *(Refutational Completeness) Let S be a saturated set of clauses. S is unsatisfiable iff $\Box \in S$.*

Proof If S is saturated, then by Lemma 34, S_{inst} is saturated. Moreover since S is unsatisfiable, by the Herbrand theorem S_{inst} is also unsatisfiable. By Lemma 32, $\Box \in S_{inst}$. Thus $\Box \in S$.

■

7 Constructing Saturated Clause Sets

In this section, we present a concrete algorithm to compute efficiently saturated clause sets (applying randomly the inference rules would of course be very inefficient).

We divide the clause set at hand into two parts: the *active* part and the *passive* part. Initially, all clauses are active. At each step, an active clause is chosen (the so-called "given clause") and shifted from the active set to the passive set. Then, all the resolvents of this clause with a passive clause (including the given clause itself) are computed, together with the factors of the obtained clause sets. These clauses are added to the active set. The process is iterated using another given clause until the empty clause has been generated or until the set of passive clauses is empty.

This algorithm is complete, in the sense that the whole set of generated clauses is saturated (of course this set is infinite in general), provided that the given clauses are chosen in a fair way, i.e. that no clause can stay in the active set forever (all the clauses in the active set are eventually chosen as the given clause). The simplest way to be sure that this property holds is to consider the passive list as a queue, with a FIFO policy (First In First Out). Other, more sophisticated, choice strategies can be used, for instance it is possible to select as a given clause the clause with the smallest number of symbols (in order to promote simpler inferences).

In order to reduce the search space, the redundancy rules are applied as soon as possible in order to delete useless clauses.

We use the following functions:

- *NonRedundant*(S) is the set of clauses $C \in S$ that are not strictly redundant in S.
- *Simplify*(S, S') is the set of clauses $C \in S$ that are not strictly redundant w.r.t. S'.
- *Resolvents*(C, S) is the set of clauses produced by applying the Resolution rule between the clause C and a clause in S (in one step).
- *Factors*(S) denotes the smallest set of clauses containing S and stable by the Factorisation rule (i.e. the set of clauses that are obtained by applying recursively the Factoring rule on S and on clauses in *Factors*(S)). This set is obviously finite, since the Factorisation rule decreases the number of literals in a clause.

Algorithm 1 A Resolution-based Theorem Prover

Require: S is a (finite) set of clauses
  $active \leftarrow Factors(NonRedundant(S))$
  $passive \leftarrow \emptyset$
  while $active \neq \emptyset \wedge \square \notin active$ **do**
    Choose a clause $given\_cl \in active$
    $active \leftarrow active \setminus \{given\_cl\}$
    $passive \leftarrow passive \cup \{given\_cl\}$
    $new\_cl \leftarrow Factors(Resolvents(given\_cl, passive))$
    $new\_cl \leftarrow NonRedundant(new\_cl)$
    $new\_cl \leftarrow Simplify(new\_cl, passive)$
    $new\_cl \leftarrow Simplify(new\_cl, active)$
    $active \leftarrow Simplify(active, new\_cl)$
    $passive \leftarrow Simplify(passive, new\_cl)$
    $active := active \cup new\_cl$
  end while
  if $\square \in active$ **then**
    $statut \leftarrow unsat$
  else
    $statut \leftarrow sat$
  end if
  return *statut*

This algorithm can easily be completed in order to return – in case S is unsatisfiable – the whole derivation leading to the empty clauses (as a proof of the unsatisfiability of S). To this aim, it suffices to attach to each generated clause the list of its premises and the inference rule that has been applied.

8 Complexity of the Resolution Calculus

A formula is said to be *propositional* if all the atoms occurring in it are propositional variables (then there is no quantifier and no variable). The problem of checking whether a set of propositional clauses (or a set of propositional formulae) is satisfiable or not is called SAT. The truth value of a propositional formula only depends on the truth value of the propositional variables in it, thus there is only 2^n possible interpretations, where n denotes the number of propositional variables. Thus SAT can be solved in exponential time.

Obviously, the same holds if the considered formula contains no variable and no quantifier (set of ground clauses). The complexity is 2^n where n denotes the number of atoms.

It is known that all NP problems (i.e. all problems that can be solved in polynomial time using a non deterministic algorithm) can be reduced to SAT (this has been proven by Stephen Cook in [4] and approximatively in the same time by Leonid Levin).

The exact complexity of SAT is not known (this is a very important open problem), but it is usually conjectured to be non polynomial ($P \neq NP$). In this section, we identify some classes of (ground) clauses on which the Resolution calculus is efficient in the sense that it generates only a polynomial number of clauses.

A clause is said to be *Krom* if it contains at most 2 literals.

Theorem 36 *Let S be a set of ground Krom clauses. The number of distinct non valid and non empty clauses that can be generated from S by the Resolution calculus is at most $2 \times n^2$ where n is the number of atoms in S.*

Proof Obviously, the only clauses that can be obtained by Resolution or Factorisation from clauses of length at most 2 are also of length ≤ 2. Moreover, there is only $2 \times n$ clause of length 1 (n positive literals and n negative literals), and $n \times (2n - 2)$ clauses of length 2 not containing two literals with the same atom. ■

This implies that the Resolution calculus decides the class of Krom propositional clauses in polynomial time.

A clause is said to be *Horn* if it contains at most a positive literal. A set of clauses S is Horn if any clause in S is Horn.

Let sel^+ be a selection function s.t. for any clause C containing at least a negative literal, $sel^+(C)$ only contains one negative literal in C. sel^+ is often called a *positive selection function* and the corresponding Resolution strategy is called *Positive Resolution*.

Theorem 37 *Let S be a Horn set of ground clauses. Then the number of distinct clauses that can be generated from S by the sel^+-Resolution calculus is at most $\Sigma_{C \in S} neg(C)$, where $neg(C)$ denotes the number of negative literals occurring in the clause C.*

Proof Let $L \vee C$ and $\neg L \vee D$ be two clauses in S on which the Resolution rule can be applied. Obviously, L must be selected thus C contains no negative

literal. Since $L \vee C$ is Horn, C is empty. Thus the resolvent is D. Therefore, the application of the Resolution rule on S has the effect of removing the (unique) selected negative literal in a clause. The Factorisation has a similar effect (removing an occurrence of a negative selected literal).

Since only one negative literal may be selected in each clause, only $neg(C)$ may be removed from each clause C. Therefore the total number of clauses that can be obtained is $\Sigma_{C \in S} neg(C)$. ■

The two previous results do not extend to the non ground case: it is not difficult to see that the satisfiability problem is undecidable for non ground sets of Krom or Horn clauses (even for clauses that are *both* Horn and Krom). Actually 3 clauses only are sufficient to get an undecidable problem: one binary Horn clause with a positive literal and a negative one, and two unit clauses (one positive and one negative) [7].

9 Termination of the Resolution Calculus

Clearly, the Resolution calculus does not terminate in general. The set of clauses that can be deduced from a given set of clauses may be infinite (as we shall see in the last part of the course, first order logic is not decidable). For instance the reader can check that the Resolution rule applied with a positive selection strategy on $S = \{P(a), \neg P(x) \vee P(f(x))\}$ generates an infinite number of clauses of the form $P(f^n(a))$, where $n \in \mathbb{N}$. Of course S is satisfiable (constructing a model of S is a trivial exercise).

Sometimes, termination can be ensured by chosen an appropriate selection strategy. For instance, if one select the literal $P(f(x))$ in the second clause of S (instead of selected $\neg P(x)$) then no Resolution inferences are applicable hence the calculus terminates. This can be done by orienting the atoms in such a way that we have $P(x) < P(f(x))$, and by selecting the maximal literal in each clause.

There exist some syntactic classes of sets of clauses for which the Resolution calculus terminates (i.e. generates only a finite number of distinct clauses). This implies that the calculus is a decision procedure for these classes (that are of course less expressive than full first order logic). A trivial example is the Herbrand class (set of unit clauses): in this case the only resolvent that can be generated in the empty clause (thus either $\square$ can be derived in one step or the set of clauses is satisfiable).

In this section, we provide other, more interesting, examples:

9.1 Horn clauses without Functions

Theorem 38 *Let S be a set of Horn clauses containing no function symbol. The number of clauses that can be generated from S using a positive selection function sel^+ is finite (up to a renaming of variables).*

Proof We have shown (see Theorem 37) that the sel^+-Resolution rule can only be applied if one of the premises is a unit positive clause. The obtained

clause is simply obtained by removing one of the negative literals from the other premise and by applying the m.g.u.. Thus the resolvent is a Horn clause, and its length is smaller than the one of the premises. The same holds for the Factorisation rule.

Consequently, the length of the generated clauses is bounded. Since the clauses contain no function symbols, there can be only a finite number of clauses of a fixed length (up to a renaming of variables). ■

Unfortunately the previous result does not hold if the clauses are non Horn. For instance, let us consider the set of clauses $S = \{q(x,y) \vee p(x,y), \neg p(u,v) \vee p(u,w) \vee p(w,v)\}$. It is easy to check that for all $n = \mathbb{N}$, we have:

$$S \vdash_{sel^+} q(x_1, x_n) \vee \bigvee_{i=1}^{n-1} p(x_i, x_{i+1}),$$

where $x_1, \ldots, x_n$ are distinct variables (this can be proven by an easy induction on n).

9.2 Monadic Formulae

Definition 39 *A formula is said to be* monadic *iff it contains no constant/function symbol and if all the predicate symbols are of arity* 1.

In this section, we provide a selection function ensuring termination of the Resolution calculus on any set of clauses obtained from a monadic formula. We assume that the formula is in *prenex form*, i.e. of the form $Q_1 x_1, \ldots, Q_1 x_n \phi$, where $Q_1, \ldots, Q_n$ are quantifiers ($\exists$ or $\forall$) and ϕ is quantifier-free (this is clearly not restrictive because any formula can be transformed into an equivalent formula in prenex form).

We firstly introduce some notations.

Let t, s be two terms. We write $t \preceq s$ if one of the following holds:

- s is of the form $f(s_1, \ldots, s_n)$ for some function symbol f and $s_i = t$ for some $i \in [1..n]$,
- or if t, s are respectively of the form $f(t_1, \ldots, t_n)$ and $g(t_1, \ldots, t_m)$ where $m \geq n$ (we may have $f = g$).

The reader can easily check that $\preceq$ is transitive. We write $t \prec s$ iff $t \preceq s$ and $s \not\preceq s$. $\prec$ is a strict ordering. We write $t \sim s$ if $t \preceq s$ and $s \preceq t$.

Proposition 40 *Let L, L' be two literals and let σ be a substitution. If $L \preceq L'$ then $L\sigma \preceq L'\sigma$.*

Proof Immediate. ■

If $p(t)$ and $q(s)$ are two monadic atoms, we write $p(t) \preceq q(s)$ (resp. $p(t) \prec q(s)$) iff $t \preceq s$ (resp. $t \prec s$). This relation is extended to literals by ignoring the negation symbol.

We first analyse the clauses occurring in the clausal form of a monadic prenex formulae, and we show that they fulfill some particular useful properties.

Definition 41 *A clause C is said to be* regular *w.r.t. a vector of variables $x_1, \ldots, x_n$ if all the atoms in L are of the form $p(t)$ where t is:*

- *Either a variable x_i where $i \in [1..n]$.*
- *Or of the form $f(x_1, \ldots, x_m)$ for some $m \in [1..n]$.*

Lemma 42 *If ϕ is a monadic formula in prenex form and S is a clausal form of ϕ then all clauses in S are regular.*

Proof ϕ is of the form $Q_1 x_1, \ldots, Q_i x_n \psi$, where ψ is quantifier free and contains no function symbols and no constant. Let $y_1, \ldots, y_m = x_{i_1}, \ldots, x_{i_m}$ the subsequence of $x_1, \ldots, x_n$ s.t. $Q_{i_j} = \forall$, for every $j \in [1..m]$.

The atoms in the clausal form of ϕ are obtained from atoms in ϕ by skolemisation (skolemisation is the only rule that can affect atoms, the remaining rules only affect the logical part of the formula). All the atoms in ϕ are of the form $p(x_i)$ for some $i \in [1..n]$. If x_i occurs in $y_1, \ldots, y_m$ then $p(x_i)$ is not affected by skolemisation. Otherwise x_i is replaced by a skolem term of the form $f(y_1, \ldots, y_k)$ for some $k \in [1..m]$ (where k is the greatest index s.t. $i_k < i$).

Thus all the clauses in ϕ are regular w.t.t. $y_1, \ldots, y_m$. ■

We assume that the ordering $<$ satisfies the following property: if $p(t) \prec q(s)$ then $p(t) < q(s)$. This implies that if L is maximal in C, then for every L' occurring in C, we have $L \not< L'$. For every clause C, we define $sel(C)$ as the set of maximal literals in C.

The following definition and lemma state an interesting property of the maximal literals occurring in a regular clause.

Definition 43 *A clause C is said to be* decomposable *iff it is of the form $C = C_1 \vee C_2$ where C_1, C_2 are non empty and where $Var(C_1) \cap Var(C_2) = \emptyset$.*

For instance $p(x) \vee q(y) \vee r(x)$ is decomposable (with $C_1 = p(x) \vee r(x)$ and $C_2 = q(y)$) but $p(x) \vee q(y) \vee r(x, y)$ is not.

Proposition 44 *Let C be a regular, non decomposable clause.*
If L is maximal in C, then $Var(C) \subseteq Var(L)$.

Proof C is regular w.r.t. a sequence of variables $x_1, \ldots, x_n$. We assume, w.l.o.g., that the variables $x_1, \ldots, x_n$ occur in C (the variables not occurring in C may be simply deleted from the sequence $x_1, \ldots, x_n$).

By definition, the atoms in C are either of the form $p(x_i)$ for some $i \in [1..n]$ or of the form $p(f(x_1, \ldots, x_m))$ where $m \in [1..n]$.

Let $j \in [1..n]$ be an index s.t. x_j does not occur in a complex term in C. Clearly, all the atoms containing x_j must of be form $q(x_j)$ for some predicate

symbol q. Let C_1 be the disjunction of the literals in C that are of the form $q(x_j)$ or $\neg q(x_j)$ and let C_2 be the disjunction of the remaining literals.

By definition $Var(C_1) \cap Var(C_2) = \emptyset$ (C_1 only contains the variable x_j and C_2 does not contain x_j). Since C is non decomposable and $C_1 \neq \Box$, we must have $C_2 = \Box$, thus $C = C_1$. Therefore, x_j is the unique variable in C (we have $j = 1 = n$) and $Var(L) = Var(C) = \{x_j\}$.

Now, assume that all the variables in $x_1, \ldots, x_n$ occur in a complex term. This means that there must exists in $L \vee C$ an atom of the form $p(f(x_1, \ldots, x_n))$, where p is a predicate symbol and f a function symbol. Since L is maximal, we have $L \not\prec p(f(x_1, \ldots, x_n))$, thus the atom in L must also be of the form $q(g(x_1, \ldots, x_n))$ for some predicate symbol q and some function symbol g (indeed if the atom is of the form $q(x_i)$ for some $i \in [1..n]$ of $q(g(x_1, \ldots, x_m))$ for some $m < n$, we would have $L \prec p(f(x_1, \ldots, x_n))$ hence $L < p(f(x_1, \ldots, x_n))$.

Therefore, L contains all the variables in C.

■

Roughly speaking, the idea of the proof could be summarized as follows:

- We show that the class of regular clause is stable by Resolution and Factorisation (i.e. that if S is regular, then $D_{sel}(S)$ is also regular).
- We show that the number of distinct regular clauses is finite (up to a renaming of variables).

These two points together ensure termination. Unfortunately, the first point does not hold !

Consider for example the clauses $p(f(x)) \vee \neg q(y)$ and $q(g(u)) \vee q(v)$. Both clauses are regular (w.r.t. x, y and u, v respectively) but the resolvent $p(f(x)) \vee q(g(u))$ is not. However, the resolvent can be "decomposed" into a disjunction of variable-disjoint clauses $p(f(x))$ and $q(g(u))$ that are both regular.

Therefore, we need to replace the notion of regularity by a weaker notion, called "weak regularity".

Definition 45 *A clause is said to be* weakly regular *iff it is of the form* $\bigvee_{i=1}^{n} C_i$ *where the* $C_1, \ldots, C_n$ *are regular and non decomposable and for every pair* $(i, j) \in [1..n]^2$*, if* $i \neq j$ *then* $Var(C_i) \cap Var(C_j) = \emptyset$ *(the* C_i*'s share no variable).*

Proposition 46 *Every regular clause is also weakly regular.*

Proof This is obvious, since any clause can be decomposed into a disjunction of variable-disjoint clauses and since every subclause of a regular clause is itself regular, by definition. ■

Then next lemma shows that the class of weakly regular clauses is stable by Resolution and Factorisation.

Lemma 47 *If* S *is weakly regular, then* $D_{sel}(S)$ *is weakly regular.*

Proof Let $C \in D_{sel}(S)$. Assume that C is obtained by Resolution from two clauses D and E in S. By definition, D, E are of the form $p(t) \vee D'$ and $\neg p(s) \vee E'$, where σ is the m.g.u. of t and s and $C = (D' \vee E')\sigma$.

Since D, E are weakly regular, D' and E' are respectively of the form $D_1 \vee D_2$ and $E_1 \vee E_2$ where $p(t) \vee D_1$ and $\neg p(s) \vee E_1$ are regular (w.r.t. to sets of variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ respectively) and non decomposable, and where $Var(p(t) \vee D_1) \cap Var(D_2) = Var(\neg p(s) \vee E_1) \cap Var(E_2) = \emptyset$.

We have $C = D_1\sigma \vee D_2 \vee E_1\sigma \vee E_2$, and $Var(D_1\sigma \vee E_1\sigma) \cap Var(D_2 \vee E_2) = \emptyset$. $D_2 \vee E_2$ is almost regular. It suffices to show that $(D_1 \vee E_1)\sigma$ is regular.

We distinguish several cases.

- If both t and s are variables: $t = x$ and $s = y$. σ is the substitution $\{y \mapsto x\}$ (or equivalently $\{x \mapsto y\}$). Since $p(t)$ and $q(s)$ are maximal, by Proposition 44, all the atoms in D_1 are of the form $q(x)$ and all the atoms in E_1 are of the form $q'(y)$. Consequently, $(D_1 \vee E_1)\sigma$ only contains atoms of the form $q(x)$ (or $q(y)$), thus is regular.

- If t is a variable x and s is of the form $f(y_1, \ldots, y_m)$. All the atoms in D_1 are of the form $q(x)$.

  In this case, σ is $\{x \mapsto s\}$. Thus we have $E_1\sigma = E_1$. Moreover, all the atoms in $D_1\sigma$ are of the form $q(s)$. Clearly, $D_1\sigma \vee E_1$ is regular w.r.t. $y_1, \ldots, y_m$.

- The proof is similar if s is a variable and t a complex term.

- If t, s are both complex, then we must have $t = f(x_1, \ldots, x_i)$ and $s = f(y_1, \ldots, y_j)$. The m.g.u. of t, s is of the form $\{y_i \mapsto x_i \mid i \in [1..n]\}$ (up to a renaming).

  Moreover since $p(t)$ and $\neg p(s)$ are both maximal in their clause we must have $i = n$ and $j = m$. All the variables in E_1 occurs in $y_1, \ldots, y_m$, thus $E_2\sigma$ is regular w.r.t. $x_1, \ldots, x_n$ and $D_1 \vee E_1\sigma$ is also regular.

The proof for Factorisation is similar. ■

Unfortunately, this do not give the desired result, since there is an infinite number of weakly regular clauses. Indeed, the number of variables is not bounded hence we can "repeat" different renamings of the same clause indefinitely. For instance $\bigvee_{i=1}^{n} p(x_i)$ is weakly regular for any n.

However, if we closely inspect this clause, it is clear that the Factorisation rule may be applied on the literals $p(x_n)$ and $p(x_{n-1})$. This produces the clause $\bigvee_{i=1}^{n-1} p(x_i)$, which subsumes $\bigvee_{i=1}^{n} p(x_i)$.

If we assume that the Factorisation rule is applied in a systematic way on each generated clause and that subsumption is used to delete redundant clauses (which is always the case in practice, see for instance the algorithm in Section 7), then the previous clause will be immediately deleted.

This shows that we do not need to consider clauses containing two variants of the same clause (up to a renaming of variables). With this proviso, it is easy to see that the number of remaining clauses is finite.

In order to formalize this idea, we need the following:

Definition 48 *A clause C is said to be* condensed*, if there is no factor of C that subsumes C.*

As we have seen, non condensed clauses are redundant.

Lemma 49 *The number of weakly regular condensed clauses is finite (up to a renaming of variables).*

Proof Any weakly regular clause can be decomposed into a disjunction $\bigvee_{i=1}^{n} C_i$, where the clauses are regular, non decomposable, and where $C_1, \ldots, C_n$ share no variables.

By definition of the notion of regular clauses, each clause C_i contains at most m variables, where m is the greater arity of the function symbol in f (min. 1). Indeed, if a variable x in C_i does not occur on the scope of a function symbol then it must be the only variable in the clause (otherwise the clause could be decomposed by isolating all the variables containing x).

If there exists i, j s.t. C_i and C_j are identical up to renaming of variables, then clearly the Factorisation rule can be applied on the literals in C_i, C_j and generates the clause $\bigvee_{k \in [1..n] \setminus j} C_k$. This clause subsumes the clause $\bigvee_{i=1}^{n} C_i$. Since the clause is condensed, this is impossible.

Thus we assume that all the C_i's are distinct (up to a renaming).

The depth of the C_i is at most 2. Consequently the number of distinct C_i is bounded (up to a renaming). Thus the number of distinct weakly regular clauses is bounded. ■

The previous results shows that the Resolution calculus is a decision procedure for the monadic class (with the selection function above).

Splitting

Another way to get rid of decomposable clause that deserves to be mentioned is to apply a *splitting rule* on sets of clauses. The following lemma shows that this is possible:

Lemma 50 *Let S be a set of clauses. Let $C \vee D$ be a clause in S s.t. $Var(C) \cap Var(D) = \emptyset$. S is unsatisfiable iff $S \cup \{C\}$ and $S \cup \{D\}$ are both unsatisfiable.*

Proof If $S \cup \{C\}$ or $S \cup \{D\}$ is satisfiable then S is obviously satisfiable. Assume that S has a model I. Assume that $I \not\models S \cup \{C\}_{inst}$ and that $I \not\models S \cup \{D\}_{inst}$. Since $I \models S$, we have $I \models S_{inst}$, thus there exists two ground substitutions σ and θ of domains $Var(C)$ and $Var(D)$ respectively s.t. $I \not\models C\sigma$ and $I \not\models D\theta$. Since $Var(C)$ and $Var(D)$, we can define the substitution $\eta \stackrel{\text{def}}{=} \sigma \cup \theta$. η is a ground substitution of $Var(C \vee D)$ thus $(C \vee D)\eta \in S_{inst}$. We have $I \not\models (C \vee D)\eta$, which is impossible since $I \models S_{inst}$.

Thus at least one of the set $S \cup \{C\}_{inst}$ or $S \cup \{D\}_{inst}$ is satisfiable, which implies by the Herbrand theorem that either $S \cup \{C\}$ or $S \cup \{D\}$ is satisfiable. ■

Using this lemma, a set of clauses containing a decomposable clause may be replaced by two clause sets which have to be refuted separately (by a recursive call to the theorem prover) in order to show that the original clause set is unsatisfiable. Refutational completeness is preserved, since (intuitively) the number of splitting rules that can be applied on a given clause set is finite. The interested reader can consult [11] for a more sophisticated version of the splitting rule.

10 Handling Equality

In the second part of the course, efficient techniques for handling equational reasoning are introduced, based of rewriting. In this section, we simply show that the satisfiability problem for a set of clauses with equality can be reduced to the non equational satisfiability problem. This result allows us to use the Resolution calculus as a semi decision procedure for equational set of clauses.

From a syntactic point of view, equality is a binary predicate symbol, usually written in infix notation: $t = s$. It is often denoted by $\approx$ (in order to avoid confusion with semantic equality).

Semantically, a formula of the form $t = s$ should hold if and only if t and s have the same value in the considered interpretation. This is formalised by the following:

Definition 51 *(E-Interpretation) An interpretation is said to be a E-*interpretation *if for any pair of elements v, v', $v =_I v'$ is true iff $v = v'$.*

The notion of E-model, E-satisfiability,... are defined accordingly.

This definition is fully satisfactory from a purely semantic point of view but it has an important disadvantage: if we replace the notion of satisfiability by the stronger notion of E-satisfiability, our previous completeness result is no more valid. For instance $S = \{a = b, b = c, a \neq c\}$ is clearly E-unsatisfiable (if a, b, c are constant symbols), but the Resolution rule cannot be applied on S thus $\square$ cannot be deduced.

This can be overcome by designing an additional inference rule for handling equality, which essentially replaces equals by equals inside a term. This rule is called *paramodulation* (or *superposition*). The reader can refer to [2] for details.

In this section, we show another idea, namely to add **additional axioms** in order to encode the properties of the equality symbol. This is a "lazy" way of handling the equality predicate, since one does not have to change the calculus (and more importantly, the implementation !).

Obviously equality has the following properties:

- It is **reflexive**, i.e. $t = t$, for any term t.
- If is **commutative**, i.e. $t = s$ iff $s = t$.
- It is **transitive**: if $t = s$ and $s = u$ then $t = u$.

- If has the **substitutivity property**: if one replace, in a given term or formula, a subterm t by a term s s.t. $t = s$, then the value of the term or formula is not affected.

These properties can be easily expressed in first-order logic. We denote by EQ the following set of axioms (x, y, z denotes variables):

$x = x$	Reflexivity
$x = y \vee y \neq x$	Commutativity
$x \neq y \vee y \neq z \vee x = z$	Transitivity
$\bigvee_{i=1}^{n} x_i \neq y_i \vee f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$	Substitutivity
f is a function symbol.	
$\bigvee_{i=1}^{n} x_i \neq y_i \vee \neg P(x_1, \ldots, x_n) \vee P(y_1, \ldots, y_n)$	Substitutivity
P is a predicate symbol.	

EQ may be infinite in principle, since there may be an infinite number of function or predicate symbols (since we use first-order logic, we cannot quantify over functions or predicates). However, one can limit oneself to the symbols actually occurring in the given formula, in order to obtain a finite set of clauses.

It is easy to add EQ to the considered clause sets before applying the Resolution proof procedure. However, a problem remains: the above properties do not ensure that $=_I$ is the identity, but merely that it is a *congruence* on the considered language. For instance, let us consider the following interpretation (on the language containing only the predicate symbol P and the function f):

$$\begin{array}{lcl} D_I & \stackrel{\text{def}}{=} & \mathbb{N} \\ P_I(x) & = & \textit{true} \text{ iff } x \text{ is even} \\ x =_I y & = & \textit{true} \text{ if } x + y \text{ is even} \\ f_I(x) & = & x + 1 \end{array}$$

The reader can check $I \models EQ$. However, I is not a E-interpretation.

Fortunately the next theorem states that any model satisfying EQ can be reduced into a E-model, thus ensuring that satisfiability is preserved.

Theorem 52 *Let I be an interpretation satisfying EQ. There exists a E-interpretation I' s.t. for any formula ϕ, $I \models \phi$ iff $I' \models \phi$.*

Proof Obviously, $=_I$ is an equivalence relation. We denote by $\overline{v}$ the equivalence class of any element $v \in D_I$.

We define the interpretation I' as follows:

- $D_{I'} \stackrel{\text{def}}{=} \{\overline{v} \mid v \in D_I\}$. The domain of I' is the set of equivalence classes of the domain of I.
- For any constant symbol (or variable) a, $a_{I'} \stackrel{\text{def}}{=} \overline{a_I}$.
- For any propositional variable P, $P_{I'} \stackrel{\text{def}}{=} P_I$.

- For any function symbol f of arity n and for any n-tuple $(v_1, \ldots, v_n) \in D_{I'}^n$: $f_{I'}(v_1, \ldots, v_n) \stackrel{\text{def}}{=} \overline{f_I(v'_1, \ldots, v'_n)}$, where for every $i \in [1..n]$, v'_i is an element (arbitrarily chosen) occurring in the equivalence class v_i.
- For any predicate symbol P of arity n and for any n-tuple $(v_1, \ldots, v_n) \in D_{I'}^n$: $P_{I'}(v_1, \ldots, v_n) \stackrel{\text{def}}{=} P_I(v'_1, \ldots, v'_n)$, where for every $i \in [1..n]$, v'_i is an element (arbitrarily chosen) occurring in the equivalence class v_i.

We show that for any term t and any formula ϕ, the following relations hold: $[t]_{I'} = \overline{[t]_I}$ and $[\phi]_{I'} = [\phi]_I$.

The proof is by induction on the size of t and ϕ.

Terms:

- If t is a constant or a variable, then the property follows immediately from the definition.
- Assume that t is a complex term of the form $f(t_1, \ldots, t_n)$. By the induction hypothesis, we have $\forall i \in [1..n], [t_i]_{I'} = \overline{[t_i]_I}$. Moreover, by definition of I', we have $[t]_{I'} = \overline{f_I(v'_1, \ldots, v'_n)}$ where $\forall i \in [1..n], v'_i \in [t_i]_{I'}$. Thus $v'_i =_I [t_i]_I$. By the substitutivity axiom we deduce that $[t] =_I f_I(v'_1, \ldots, v'_n)$. Thus $[t]_{I'} = \overline{[t]_I}$.

Formulae:

- If ϕ is a propositional variable then the proof follows directly from the definition of I'.
- Assume that ϕ is an atom of the form $t = s$. Then we have $[\phi]_I = \mathit{true}$ iff $[t] =_I [t]_I$, i.e. iff $\overline{[t]_I} = \overline{[s]_I}$ i.e. iff $[t]_{I'} = [s]_{I'}$.
- Assume that ϕ is an atom of the form $P(t_1, \ldots, t_n)$. By the induction hypothesis, we have $\forall i \in [1..n], [t_i]_{I'} = \overline{[t_i]_I}$. Moreover, by definition of I', we have $[t]_{I'} = P_I(v'_1, \ldots, v'_n)$ where $\forall i \in [1..n], v'_i \in [t_i]_{I'}$. Thus $v'_i =_I [t_i]_I$. By the substitutivity axiom we deduce that $[t]_I = P_I(v'_1, \ldots, v'_n)$. Thus $[t]_{I'} = [t]_I$.
- If ϕ is of the form $\neg\psi$, then by induction we have $[\psi]_{I'} = [\psi]_I$. Thus $[\phi]_{I'} = \neg[\psi]_{I'} = \neg[\psi]_I = [\neg\psi]_I = [\phi]_I$.
- If ϕ is of the form $\psi_1 \star \psi_2$ where $\star$ is a logical connective $\vee, \wedge, \Leftrightarrow, \Rightarrow$ then by induction we have $[\psi_i]_{I'} = [\psi_i]_I$ $(i = 1, 2)$. Thus $[\phi]_{I'} = [\psi_1]_{I'} \star [\psi_2]_{I'} = [\psi_1]_I \star [\psi_2]_I = [\psi_1 \star \psi_2]_I = [\phi]_I$.
- If ϕ is of the form $\forall x\psi$, then by induction we have $[\psi]_{I\{x \leftarrow v\}'} = [\psi]_{I\{x \leftarrow v\}}$. Moreover, it is clear, by definition of I', that $I\{x \leftarrow v\}' = I'\{x \leftarrow \overline{v}\}$. $[\phi]_I$ is *true* iff for all $v \in D_I$ we have $[\phi]_{I\{x \leftarrow v\}} = \mathit{true}$, i.e. $[\phi]_{I'\{x \leftarrow \overline{v}\}} = \mathit{true}$. All elements in $D_{I'}$ are of the form $\overline{v}$ for some $v \in D_I$. Thus we have $\forall v \in D_I, [\phi]_{I'\{x \leftarrow \overline{v}\}} = \mathit{true}$ iff $\forall v \in D_{I'}, [\phi]_{I'\{x \leftarrow v\}} = \mathit{true}$ i.e. iff $[\phi]_{I'} = \mathit{true}$.
- The proof is similar if $\phi = \exists x\psi$.

■

Corollary 53 *For any set of clauses S, S is E-unsatisfiable iff $S \cup EQ$ is unsatisfiable.*

Proof If S is E-satisfiable then obviously $S \cup EQ$ is satisfiable since the identity satisfies EQ. If $S \cup EQ$ is satisfiable, then there exists an interpretation I satisfying S and EQ. By Theorem 52, there exists a E-interpretation I' satisfying S. ■

Although the previous result allows to use to Resolution calculus to handle the equality predicate, from a practical point of view, it is more efficient to use specific inference rules (which are based on the substitutivity property: replacing equals by equals).

References

[1] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[2] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[3] T. Boy de la Tour. An optimality result for clause form translation. *Journal of Symbolic Computation*, 14:283–301, 1992.

[4] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[5] R. David, K. Nour, and C. Raffalli. *Introduction la logique. Thorie de la dmonstration*. Dunod, 2004.

[6] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[7] P. Hanschke and J. Würtz. Satisfiability of the smallest binary program. *IPL*, 45(5):237–241, 1993.

[8] A. Leitsch. *The resolution calculus*. Springer. Texts in Theoretical Computer Science, 1997.

[9] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Form. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science, 2001.

[10] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[11] A. Riazanov and A. Voronkov. Splitting without backtracking. In B. Nebel, editor, *17th International Joint Conference on Artificial Intelligence*, pages 611–617. Morgan Kaufman, 2001.

[12] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning.* North-Holland, 2001.

[13] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12:23–41, 1965.