Applications of Boolean and First-order Logic

Mikko Malinen

31st August, 2005

Abstract

This article presents applications of Boolean and first-order logic, i.e. how logic is used in solving certain problems. Some examples list an input file to theorem prover Otter. Prerequisite knowledge of Boolean and first-order logic is assumed, although the languages of Boolean and first-order logic are defined.

1 The Language of Propositional Logic (Boolean Logic)

This section is from [1]:

Propositional formulae (or propositions) are strings of symbols from a countable alphabet defined below, and formed according to certain rules stated in definition (1.2).

Definition 1.1 *(The alphabet for propositional formulae) This alphabet consists of:*
(1) A countable set $\boldsymbol{PS}$ *of proposition symbols:* $P_0, P_1, P_2...$*;*
(2) The logical commectives: $\wedge$ *(and),*$\vee$ *(or),*$\Rightarrow$ *(implication),* $\neg$ *(not) and sometimes* $\Leftrightarrow$ *(equivalence) and the constant* $\perp$ *(false);*
(3) Auxiliary symbols: "(" (left parenthesis), ")" (right parenthesis).

The set *PROP* of propositional formulae (or propositions) is defined as the inductive closure of a certain subset of the alphabet of definition (1.1) under certain operations defined below.

Definition 1.2 *Propositional formulae. The set PROP or propositional formulae (or propositions) is the inductive closure of the set* $\boldsymbol{PS} \cup \{\perp\}$ *under the functions* $C_\neg$,$C_\wedge$,$C_\vee$,$C_\Rightarrow$ *and* $C_\Leftrightarrow$*, defined as follows: For any two strings A,B*

*over the alphabet of definition (*1.1*),*

$$C_{\neg}(A) = \neg A,$$
$$C_{\wedge}(A, B) = (A \wedge B),$$
$$C_{\vee}(A, B) = (A \vee B),$$
$$C_{\Rightarrow}(A, B) = (A \Rightarrow B) \quad \textit{and}$$
$$C_{\Leftrightarrow}(A, B) = (A \Leftrightarrow B).$$

The above definition is the official definition of *PROP* as an inductive closure, but is a bit formal. For that reason, it is often stated less formally as follows:
The set *PROP* of propositions is the smallest set of strings over the alphabet of definition (1.1), such that:
(1) Every proposition symbol P_i is in *PROP* and $\bot$ is in *PROP*;
(2) Whenever A is in *PROP*, $\neg A$ is also in *PROP*;
(3) Whenever A,B are in *PROP*,$(A \vee B)$,$(A \wedge B)$,$(A \Rightarrow B)$ and $(A \Leftrightarrow B)$ are also in *PROP*;
(4) A string is in *PROP* only if it is formed by applying the rules (1),(2),(3).

2 First-order Languages

This section is almost entirely from [3]:

We recall what a first-order language is, a notion essentially due to G.Frege. By necessity our treatment is reduced to a list of definitions. A *first-order language* consists of an alphabet and all formulas defined over it. An *alphabet* consists of the following classes of symbols:

- *variables* denoted by *x,y,z,v,u,...,*
- *constants* denoted by *a,b,c,d,...,*
- *function symbols* denoted by *f,g,...,*
- *relation symbols* denoted by *p,q,r,...* or *P,Q,R,...,*
- *propositional constants*, which are **true** and **false**,
- *connectives*, which are $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence)
- *quantifiers*, which are $\exists$ (there exists) and $\forall$ (for all),
- *parentheses*, which are (and) and the *comma*, that is:,.

Thus the sets of connectives, quantifiers and parentheses is fixed. We assume also that the set of variables is infinite and fixed. Those classes of symbols are called *logical symbols*. The other classes of symbols, that is, constants, relation symbols (or just *relations*) and function symbols (or just *functions*), may vary

and in particular may be empty. They are called *nonlogical symbols.* Each first-order language is thus determined by its nonlogical symbols.
Each function and relation symbol has a fixed *arity*, that is, the number of arguments. We assume that functions have a positive arity - the role of 0-ary functions is played by the constants. In contrast, 0-ary relations are admitted. They are called *propositional symbols*, or simply *propositions.* Note that each alphabet is uniquely determined by its constants, functions and relations.
We now define by induction two classes of strings of symbols over a given alphabet. First we define the class of *terms* as follows:

- a variable is a term
- a constant is a term
- if f is an n-ary function and $t_1, ..., t_n$ are terms then $f(t_1, ..., t_n)$ is a term

Terms are denoted by s, t, u. Finally, we define the class of *formulas* as follows:

- if p is an n-ary relation and $t_1, ..., t_n$ are terms then $p(t_1, ..., t_n)$ is a formula called *atomic formula*, or just an *atom*,
- **true** and **false** are formulas,
- if F and G are formulas then so are $\neg F, (F \wedge G), (F \vee G), (F \Rightarrow G)$ and $(F \Leftrightarrow G)$,
- if F is a formula and x is a variable then $\exists x F$ and $\forall x F$ are formulas.

Sometimes we shall write $(G \Leftarrow F)$ instead of $(F \Rightarrow G)$. Some well known binary functions (like $+$) or relations (like $=$) are usually written in *infix notation* i.e. between the arguments. Atomic formulas are denoted by A, B and formulas in general by F, G. If F is a quantifier-free formula with variables $x_1, ..., x_n$ we write $\exists F$ for $\exists x_1 ... \exists x_n F$ and $\forall F$ for $\forall x_1 ... \forall x_n F$. Formulas of the form $\forall F$ are called *universal formulas.* A term or formula with no variables is called *ground.* Given two strings of symbols e_1 and e_2 from the alphabet, we write $e_1 \equiv e_2$ when e_1 and e_2 are identical. Usually these strins will be terms or formulas.
The definition of formulas is rigorous at the expence of excessive use of parentheses. One way to eliminate most of them is by introducing a *binding order* among the connectives and quantifiers. We thus assume that $\neg, \exists$ and $\forall$ bind stronger than $\vee$ which in turn binds stronger than $\wedge$ which binds stronger than $\Rightarrow$ and $\Leftrightarrow$. Also, we assume that $\vee, \wedge, \Rightarrow$ and $\Leftrightarrow$ *associate to the right* and omit the outer parentheses. Thus, thanks to the binding order, we can rewrite the formula

$$\forall y \forall x((p(x) \wedge \neg r(y)) \Rightarrow (\neg q(x) \vee (A \vee B)))$$

as

$$\forall y \forall x(p(x) \wedge \neg r(y) \Rightarrow \neg q(x) \vee (A \vee B))$$

which, thanks to the convention of the association to the right, further simplifies to

$$\forall y \forall x(p(x) \wedge \neg r(y) \Rightarrow \neg q(x) \vee A \vee B).$$

This completes the definition of a first-order language.

3 About Proof Systems

This section is almost entirely from [1]:

Every logical system consists of a *language* used to write statements called *propositions* or *formulae*. Normally, when one writes a formula, one has some intended *interpretation* of this formula in mind. For example, a formula may assert a true property about the natural numbers, or some property that must be true in a database. This implies that a formula has a well-defined *meaning* or *semantics*. But how do we define this meaning precisely? In logic, we usually define the meaning of a formula as its *truth value*. A formula can be either true (or valid) or false.
Defining rigorously the notion of truth is actually not as obvious as it appears. We shall present a concept of truth due to Tarski. Roughly speaking, a formula is true if it is satisfied in all possible interpretations.
The next question is to investigate whether it is possible to find methods for deciding in a finite number of steps whether a formula is true (or valid). This is a very difficult task. In fact, by a theorem due to Church, there is no such general method for first-order logic.
However, there is another familiar method for testing whether a formula is true: to give a *proof* of this formula.
Of course, to be of any value, a proof system should be *sound*, which means that every provable formula is true.
The branch of logic concerned with the study of proof is known as *proof theory*.
Now, if we have a sound proof system, we know that every provable formula is true. Is the proof system strong enough that it is also possible to prove every true formula (of first-order logic) ?
A major theorem of Gödel shows that there are logical proof systems in which every true formula is provable. This is referred to as the *completeness* of the proof system.
To summarize the situation, if one is interested in algorithmic methods for testing whether of first-order logic is valid, there are two logical results of central importance: one positive (Gödel's completeness theorem), the other one negative (Church's undecidability of validity). Roughly speaking, Gödel's completeness theorem asserts that there are logical calculi in which every true formula is provable, and Church's theorem asserts that there is no decicion procedure (procedure which always terminates) for deciding whether a formula is true (valid). Hence, any algorithmic procedure for testing whether a formula is true (or equivalently, by Gödel's completeness theorem, provable in a complete system) must run forever when given certain non-true formulae as input.
Propositions are much simpler than first-order formulae. Indeed, there are algorithms for deciding truth of propositional formulae.
Different proof systems are Hilbert-style proof system [2][9][10], Gentzen-like sequent calculi [1], Tableau method [10][11] and resolution method [1][2][10][11].

4 Some Graph Theory

Definition 4.1 *A **Hamiltonian graph** is a graph with a spanning cycle, also called a **Hamiltonian cycle**.* [7]

Definition 4.2 *A **drawing** of a graph G is a function f defined on $V(G) \cup E(G)$ that assigns each vertex v a point $f(v)$ in the plane and assigns each edge with endpoints u, v a polygonal $f(u), f(v)$-curve. The images of vertices are distinct. A point in $f(e) \cap f(e')$ that is not a common endpoint is a **crossing**.* [7]

Definition 4.3 *A graph is **planar** if it has a drawing without crossings.* [7]

5 Applications of Boolean Logic

5.1 A Chess Related Problem

Description of the problem:
If a knight is placed at the lower left corner of the chess board, can all chess squares be visited using rules of movement for knight? One variation of this problem is called *Knight's Tour* problem. In knights tour every square may be visited only once. It turns out that there is a solution, i.e. all chess squares can be visited. There exists solutions to even knight's tour.
In this presentation we show how the problem may be solved with theorem prover Otter [6]. In this solution any number of visits to chess squares are allowed.
In the next subsection you may find listing of file knights.in. Here we explain the contents of this file. In section A of the file there is a line "Ra1.". This is a proposition which is interpreted as "Square a1 is reachable". Square a1 is reachable because there we put the knight initially. In section B of the file we put the definitions. In the beginning of section B there is line "Ra1 -> (Rb3 & Rc2).". This is interpreted as "if square a1 is reachable then squares b3 and c2 are reachable. This is where we write the moving possibilities from each square. The other rows are written in the same way. In section C we put the actual query (it's negation). Here we ask if all the squares are reachable.

5.1.1 The file knights.in

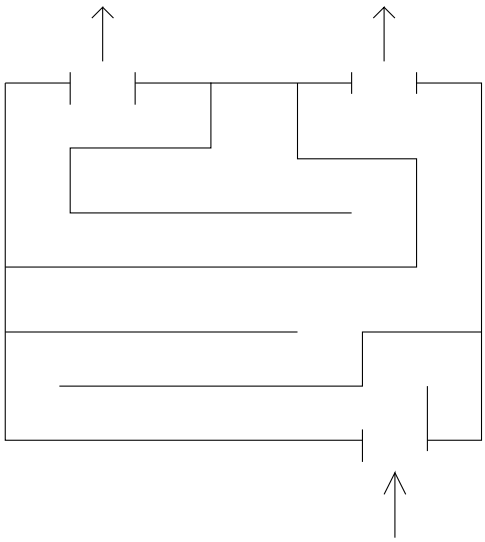
```
set(auto).
formula_list(usable).
% Section A: database
Ra1.

% Section B: definitions
Ra1 -> (Rb3 & Rc2).
Rb1 -> (Ra3 & Rc3 & Rd2).
```

```
Rc1 -> (Ra2 & Rb3 & Rd3 & Re2).
Rd1 -> (Rb2 & Rc3 & Re3 & Rf2).
Re1 -> (Rc2 & Rd3 & Rf3 & Rg2).
Rf1 -> (Rd2 & Re3 & Rg3 & Rh2).
Rg1 -> (Re2 & Rf3 & Rh3).
Rh1 -> (Rf2 & Rg3).

Ra2 -> (Rb4 & Rc3 & Rc1).
Rb2 -> (Ra4 & Rc4 & Rd3 & Rd1).
Rc2 -> (Ra1 & Ra3 & Rb4 & Rd4 & Re3 & Re1).
Rd2 -> (Rb1 & Rb3 & Rc4 & Re4 & Rf3 & Rf1).
Re2 -> (Rc1 & Rc3 & Rd4 & Rf4 & Rg3 & Rg1).
Rf2 -> (Rd1 & Rd3 & Re4 & Rg4 & Rh3 & Rh1).
Rg2 -> (Re1 & Re3 & Rf4 & Rh4).
Rh2 -> (Rf1 & Rf3 & Rg4).

Ra3 -> (Rb5 & Rc4 & Rc2 & Rb1).
Rb3 -> (Ra5 & Rc5 & Rd4 & Rd2 & Rc1 & Ra1).
Rc3 -> (Ra4 & Rb5 & Rd5 & Re4 & Re2 & Rd1 & Rb1 & Ra2).
Rd3 -> (Rb4 & Rc5 & Re5 & Rf4 & Rf2 & Re1 & Rc1 & Rb2).
Re3 -> (Rc4 & Rd5 & Rf5 & Rg4 & Rg2 & Rf1 & Rd1 & Rc2).
Rf3 -> (Rd4 & Re5 & Rg5 & Rh4 & Rh2 & Rg1 & Re1 & Rd2).
Rg3 -> (Re4 & Rf5 & Rh5 & Rh1 & Rf1 & Re2).
Rh3 -> (Rf4 & Rg5 & Rg1 & Rf2).

Ra4 -> (Rb6 & Rc5 & Rc3 & Rb2).
Rb4 -> (Ra6 & Rc6 & Rd5 & Rd3 & Rc2 & Ra2).
Rc4 -> (Ra5 & Rb6 & Rd6 & Re5 & Re3 & Rd2 & Rb2 & Ra3).
Rd4 -> (Rb5 & Rc6 & Re6 & Rf5 & Rf3 & Re2 & Rc2 & Rb3).
Re4 -> (Rc5 & Rd6 & Rf6 & Rg5 & Rg3 & Rf2 & Rd2 & Rc3).
Rf4 -> (Rd5 & Re6 & Rg6 & Rh5 & Rh3 & Rg2 & Re2 & Rd3).
Rg4 -> (Re5 & Rf6 & Rh6 & Rh2 & Rf2 & Re3).
Rh4 -> (Rf5 & Rg6 & Rg2 & Rf3).

Ra5 -> (Rb6 & Rc6 & Rc4 & Rb3).
Rb5 -> (Ra7 & Rc7 & Rd6 & Rd4 & Rc3 & Ra3).
Rc5 -> (Ra6 & Rb7 & Rd7 & Re6 & Re4 & Rd3 & Rb3 & Ra4).
Rd5 -> (Rb6 & Rc7 & Re7 & Rf6 & Rf4 & Re3 & Rc3 & Rb4).
Re5 -> (Rc6 & Rd7 & Rf7 & Rg6 & Rg4 & Rf3 & Rd3 & Rc4).
Rf5 -> (Rd6 & Re7 & Rg7 & Rh6 & Rh4 & Rg3 & Re3 & Rf4).
Rg5 -> (Re6 & Rf7 & Rh7 & Rh3 & Rf3 & Rg4).
Rh5 -> (Rf6 & Rg7 & Rg3 & Rh4).

Ra6 -> (Rb8 & Rc7 & Rc5 & Rb4).
Rb6 -> (Ra8 & Rc8 & Rd7 & Rd5 & Rc4 & Ra4).
Rc6 -> (Ra7 & Rb8 & Rd8 & Re7 & Re5 & Rd4 & Rb4 & Ra5).
```

```
Rd6 -> (Rb7 & Rc8 & Re8 & Rf7 & Rf5 & Re4 & Rc4 & Rb5).
Re6 -> (Rc7 & Rd8 & Rf8 & Rg7 & Rg5 & Rf4 & Rd4 & Rc5).
Rf6 -> (Rd7 & Re8 & Rg8 & Rh7 & Rh5 & Rg4 & Re4 & Rd5).
Rg6 -> (Re7 & Rf8 & Rh8 & Rh4 & Rf4 & Re5).
Rh6 -> (Rf7 & Rg8 & Rg4 & Rf5).

Ra7 -> (Rc8 & Rc6 & Rb5).
Rb7 -> (Rd8 & Rd6 & Rc5 & Ra5).
Rc7 -> (Ra8 & Re8 & Re6 & Rd5 & Rb5 & Ra6).
Rd7 -> (Rb8 & Rf8 & Rf6 & Re5 & Rc5 & Rb6).
Re7 -> (Rc8 & Rg8 & Rg6 & Rf5 & Rd5 & Rc6).
Rf7 -> (Rd8 & Rh8 & Rh6 & Rg5 & Re5 & Rd6).
Rg7 -> (Re8 & Rh5 & Rf5 & Re6).
Rh7 -> (Rf8 & Rg5 & Rf6).

Ra8 -> (Rc7 & Rb6).
Rb8 -> (Rd7 & Rc6 & Ra6).
Rc8 -> (Re7 & Rd6 & Rb6 & Ra7).
Rd8 -> (Rf7 & Re6 & Rc6 & Rb7).
Re8 -> (Rg7 & Rf6 & Rd6 & Rc7).
Rf8 -> (Rh7 & Rg6 & Re6 & Rd7).
Rg8 -> (Rh6 & Rf6 & Re7).
Rh8 -> (Rg6 & Rf7).

%Section C: negation of the query
-(Ra1 & Rb1 & Rc1 & Rd1 & Re1 & Rf1 & Rg1 & Rh1 & Ra2 & Rb2 & Rc2 & Rd2 & Re2
& Rf2 & Rg2 & Rh2 & Ra3 & Rb3 & Rc3 & Rd3 & Re3 & Rf3 & Rg3 & Rh3 & Ra4 & Rb4
& Rc4 & Rd4 & Re4 & Rf4 & Rg4 & Rh4 & Ra5 & Rb5 & Rc5 & Rd5 & Re5 & Rf5 & Rg5
& Rh5 & Ra6 & Rb6 & Rc6 & Rd6 & Re6 & Rf6 & Rg6 & Rh6 & Ra7 & Rb7 & Rc7 & Rd7
& Re7 & Rf7 & Rg7 & Rh7 & Ra8 & Rb8 & Rc8 & Rd8 & Re8 & Rf8 & Rg8 & Rh8).

end_of_list.
```

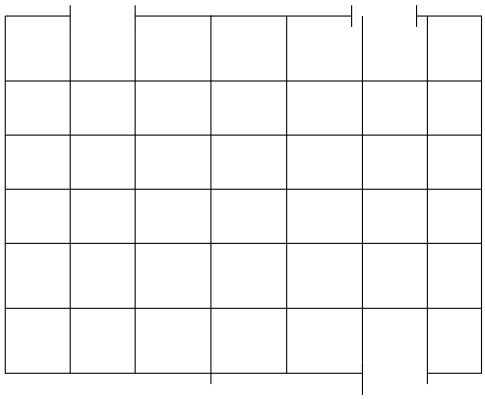
6 A Labyrinth Problem

Otter can be used to prove that there is a way out from a labyrinth. An example labyrinth is in picture 1.



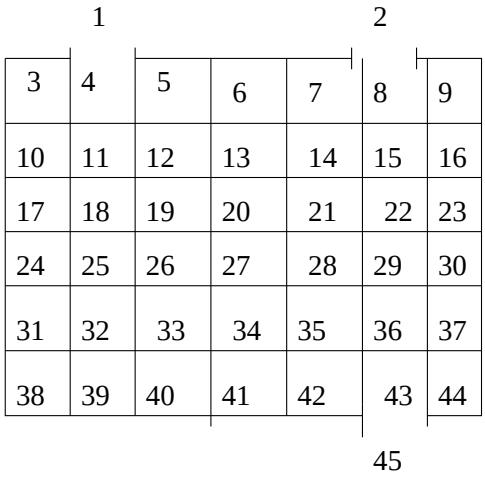
Picture 1. An example labyrinth.

The solving technique proceeds as follows.
1) Generating the squares. The walls of the squares must go along the walls of the labyrinth.



Picture 2. Generating the squares.

2) Numbering the squares



Picture 3. Numbering the squares.

3) Writing the input file. The input file labyrin.in is in the following subsection. Here we explain the contents of this file. Section A of the file has line "R45". This is interpreted as "Square (position) 45 is reachable". This is our entry position. Section B of the file has the definitions. For example line "R3 -> (R4 & R10)." is interpreted as "if square 3 is reachable then squares 4 and 10 are reachable. Here we need to define reachability only to the neighboring squares. In section C of the file we write the query (it's negation). With "-R2" we ask to prove that square (position) 2 is reachable.

6.0.2 The file labyrin.in

```
set(auto).
formula_list(usable).
% Section A: database
R45.

% Section B: definitions
R1 -> (R4).
R2 -> (R8).
R3 -> (R4 & R10).
R4 -> (R1 & R3 & R5).
R5 -> (R4).
R6 -> (R13).
R7 -> (R8).
R8 -> (R2 & R7 & R9).
R9 -> (R8 & R16).
R10 -> (R3 & R17).
```

```
R11 -> (R12).
R12 -> (R11 & R13).
R13 -> (R6 & R12 & R14).
R14 -> (R13 & R15).
R15 -> (R14 & R22).
R16 -> (R9 & R23).
R17 -> (R10 & R18).
R18 -> (R17 & R19).
R19 -> (R18 & R20).
R20 -> (R19 & R21).
R21 -> (R20 & R22).
R22 -> (R15 & R21).
R23 -> (R16 & R30).
R24 -> (R25).
R25 -> (R24 & R26).
R26 -> (R25 & R27).
R27 -> (R26 & R28).
R28 -> (R27 & R29 & R35).
R29 -> (R28 & R30).
R30 -> (R23 & R29).
R31 -> (R32 & R38).
R32 -> (R31 & R33).
R33 -> (R32 & R34).
R34 -> (R33 & R35).
R35 -> (R28 & R34).
R36 -> (R37 & R43).
R37 -> (R36 & R44).
R38 -> (R31 & R39).
R39 -> (R38 & R40).
R40 -> (R39 & R41).
R41 -> (R40 & R42).
R42 -> (R41 & R43).
R43 -> (R36 & R42).
R44 -> (R37).
R45 -> (R43).

%Section C: negation of the query
-R2.

end_of_list.
```

7 Applications of First-order Logic

7.1 About Graph Properties

Let A be a finite alphabet. We denote by $\mathbf{D}(A)$ the class of finite or infinite directed graphs, the edges of which are labelled with labels from A. Here are a few examples [4] of properties of a graph G in the class $\mathbf{D}(A)$ that are first-order expresible:

- G is simple;
- G is k-regular for some fixed integer k;
- G is of degree at most k for some fixed integer k;
- G is edge-colored by A, i.e., any two edges of G having the same label have no vertex in common.

The following properties are not expressible in first-order logic:

- G is connected
- G is planar
- G is Hamiltonian.

Theorem 7.1 *A graph property is in the class P if it is first-order.*[4]

P and other compexity classes are discussed in [8][9].

7.2 Formalizing Some Graph and Relation Properties

- Reflexivity
$$\forall x(G(x,x))$$
where $G(x,y)$ is read "there is an edge from x to y".
- Symmetricity
$$\forall x \forall y(G(x,y) \Rightarrow G(y,x))$$
- Transitivity
$$\forall x \forall y \forall z((G(x,y) \wedge G(y,z)) \Rightarrow G(x,z))$$
- The out-degree of a graph is at most 2:

  The formula $\phi(x)$ defined as:

  $\forall y_1, y_2, y_3[edg(x,y_1) \wedge edg(x,y_2) \wedge edg(x,y_3) \Rightarrow y_1 = y_2 \vee y_1 = y_3 \vee y_2 = y_3]$

  expresses that the vertex x of the represented graph has out-degree at most 2. The closed formula $\forall x \phi(x)$ expresses thus that the considered graph has outdegree at most 2.

7.3 Boolean Circuits

This application of first-order logic is from [12].

A Boolean circuit consists of a set of gates that connect the inputs of the circuit to the outputs. Three types of gates are considered here: and-gates, or-gates and inverters. These gates implement the respective logical connectives (i.e. conjunction, disjunction and negation), but gates handle Boolean values 0 and 1 rather than truth values.
Any Boolean circuit can be described using the following symbols:

- Any set of constants can be introduced as names for the input and output points. Some of these points appear as intermediary points between gates that connect outputs of gates to inputs of others.
- constants 0 and 1 refer to the two Boolean values that a point may possess.
- Predicate $Value(p, v)$: the value of a point p in the circuit is v.
- Predicate $And(x, y, z)$: there is an and-gate in the circuit that connects inputs x and y to an output z. It should be clear that the value of z is 1 only when both x and y have the value 1.
- Predicate $Or(x, y, z)$: there is an or-gate in the circuit that connects inputs x and y to an output z.
- Predicate $Inv(x, y)$: there is an inverter in the circuit that connects an input x to an output y.

Given these, our task was to write a definition for the predicate $Value$ that allows computing the value of the output of the whole circuit given the structure of the circuit as well as values of its inputs. We had to use a language based only on the symbols mentioned above.
As an example, consider a simple circuit with three gates, one of each type; i.e. an and-gate, or-gate and inverter. Suppose that the structure of this circuit is the following: the and-gate connects inputs a and b to output d, the or-gate connects inputs d and e to output f, and the inverter connects input c to output e. In terms of the given predicates, this is expressed as $And(a, b, d)$, $Or(d, e, f)$ and $Inv(c, e)$. Furthermore, suppose that the input values of the circuit are such that the point a is 0 and the points b and c are 1. This is expressed as $Value(a, 0)$, $Value(b, 1)$ and $Value(c, 1)$. Then, the definitions of the value predicate should allow us to infer that e.g. the value of the point d is 0, which is expressed as $Value(d, 0)$.
The definitions should work for arbitrary circuits (without loops) and arbitrary values for inputs.
The input file for Otter is composed of the following sections:

- Section A: description of a particular circuit and the values of inputs given in terms of predicates And,Or,Inv and $Value$.

- Section B: definition of the predicate $Value$.
- Section C: a query involving the $Value$ predicate

7.3.1 The input file for Otter

```
set(auto).
formula_list(usable).

%section A (database)

And(a,b,d).
Or(d,e,f).
Inv(c,e).
Value(a,0).
Value(b,1).
Value(c,1).


%section B (definitions)

all x y z (Value(x,1) & Value(y,1) & And(x,y,z) -> Value(z,1)).
all x y z ( (Value(x,0) | Value(y,0)) & And(x,y,z) -> Value(z,0)).
all x y z ((Value(x,1) | Value(y,1)) & Or(x,y,z) -> Value(z,1)).
all x y z ((Value(x,0) & Value(y,0)) & Or(x,y,z) -> Value(z,0)).
all x y (Value(x,0) & Inv(x,y) -> Value(y,1)).
all x y (Value(x,1) & Inv(x,y) -> Value(y,0)).


%section C (negation of the query)

-(Value(f,0)).
```

7.4 "There Is No Male Barber"

This application of first-order logic is from [13].

An assignment: Prove by resolution, that there is no male barber, when
a) Every barber shaves beards of those men, who do not shave their own beards.
b) No barber shaves beards of those men, who shave their own beards.

Solution: Let's imagine that the universum consists of a set of men. Let's use the next predicates in formalization: $P(x) =$ "x is a barber" and $A(x, y) =$"x shaves y's beard".
a) $\forall x(P(x) \Rightarrow \forall y(\neg A(y, y) \Rightarrow A(x, y)))$,
b) $\forall x(P(x) \Rightarrow \forall y(A(y, y) \Rightarrow \neg A(x, y)))$,

Let's compose the clauses:

a) $\forall x(P(x) \Rightarrow \forall y(\neg A(y,y) \Rightarrow A(x,y)))$
$\forall x(\neg P(x) \vee \forall y(A(y,y) \vee A(x,y)))$
$\forall x \forall y(\neg P(x) \vee A(y,y) \vee A(x,y))$
$\neg P(x) \vee A(y,y) \vee A(x,y)$
$\{\neg P(x_1), A(y_1,y_1), A(x_1,y_1)\}$

b) $\forall x(P(x) \Rightarrow \forall y(A(y,y) \Rightarrow \neg A(x,y)))$
$\forall x(\neg P(x) \vee \forall y(\neg A(y,y) \vee \neg A(x,y)))$
$\forall x \forall y(\neg P(x) \vee \neg A(y,y) \vee \neg A(x,y))$
$\neg P(x) \vee \neg A(y,y) \vee \neg A(x,y)$
$\{\neg P(x_2), \neg A(y_2,y_2), \neg A(x_2,y_2)\}$

We want to prove $\neg \exists x P(x)$ and therefore we compose the negation of the formula: $\exists x P(x)$. This formula is transformed to clause form $\{P(a)\}$.
From clauses $\{\neg P(x_1), A(y_1,y_1), A(x_1,y_1)\}$ and $\{\neg P(x_2), \neg A(y_2,y_2), \neg A(x_2,y_2)\}$ we get $\{\neg P(x_3)\}$ (substitution $\{x_1/x_3, x_2/x_3, y_1/x_3, y_2/x_3\}$). From clauses $\{P(a)\}$ and $\{\neg P(x_3)\}$ we get an empty clause (substitution $\{x_3/a\}$). So the set of clauses is unsatisfiable and $\neg \exists x P(x)$ follows logically from the premises.

8 References

[1] Jean H. Gallier, *Logic for Computer Science: Foundations of Automated Theorem Proving*, Harper & Row Publishers Inc.,1986

[2] Michael R. Genesereth, Nils J. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufman Publishers Inc.,1987

[3] Krzysztof R. Apt, Logic Programming, in J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volume B*, Elsevier Science Publishers,1990

[4] Bruno Courcelle, Graph Rewriting: An Algebraic and Logic Approach, in J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volume B*, Elsevier Science Publishers,1990

[5] Mikko Malinen, *Examples of Using Theorem Prover Otter*, an article published in Internet,http://users.tkk.fi/∼mmalinen, 2004

[6] *Otter: An Automated Deduction System*
http://www.mcs.anl.gov/AR/otter/

[7] Douglas B. West, *Introduction to Graph Theory*,2nd ed., Prentice-Hall Inc.,2001

[8] M.Sipser,*Introduction to the Theory of Computation*,PWS,1997

[9] Christos H.Papadimitriou,*Computational Complexity*,Addison-Wesley Publishing Company Inc.,1995

[10] Tomi Janhunen, Course *T-79.144 Logic In Computer Science: Foundations*, lecture slides (in Finnish), Helsinki University of Technology,2003

[11] Anil Nerode and Richard A.Shore,*Logic for Applications*, Second Edition, Springer-Verlag inc.,New York,1997

[12] Tomi Janhunen, Course *T-79.144 Logic In Computer Science: Foundations*, home assignment 3, Helsinki University of Technology,2003

[13] Tomi Janhunen, Course *T-79.144 Logic In Computer Science: Foundations*, exercises (in Finnish), Helsinki University of Technology,2003