Influences of Frege's Predicate Logic on Some Computational Models

Mohamad Awwad

Faculty of Arts and Sciences, American University of Beirut
(Beirut, Lebanon)
E-mail: ma304@aub.edu.lb
ORCID: 0000-0002-0699-5454

The purpose of this paper is to give some insights into the immense role of Frege's first order logic (FOL) in the development of computer science. We argue that the FOL is fundamental in computer science, and that some computer science subfields could not have existed without their theoretical foundations built on this form of logic. Among these subfields, one can mention the Type Theory, Databases, Descriptive Complexity, Artificial Intelligence, Logic Programming, and Automated Theorem Proving.

To illustrate our point, an in-depth attention will in particular be given to the foundational development of the most popular logic programming language, PROLOG, and the Automated Theorem Proving (ATP) systems.

Importantly, when studying the interactions between logic and computer science in the literature, we can observe a significant gap in the provision of the appropriate abstraction level. Specifically, we often encounter two different levels of abstraction. The first of these is relatively high even when describing technical notions in computer science, which obviously produces a lack of precision. The second adopts a technical-oriented approach which easily makes the topic and discussion unintuitive or inaccessible to the non-specialist.

The paper attempts to remedy these problems by adopting a balanced approach that provides a moderate level of abstraction that targets a deeper understanding of the topic without imposing a very technical presentation on the reader.

Keywords: Frege, Predicate Logic, First Order Logic (FOL), Computer Science, Logic Programming Language, PROLOG, Resolution Principle, Kowalski's interpretation, Automated Theorem Proving (ATP).

Received January 25, 2018; accepted March 1, 2018

Future Human Image, Volume 9, 2018:
DOI: 10.29202/fhi/9/1

Introduction

In his celebrated *Begriffsschrift* (or "concept notation") published in 1879 [Frege,1879], Gottlob Frege (1848-1925) created a system of formal logic that generate a truth-functional propositional calculus and allows, among others, the analysis of a proposition into a function

and an argument instead of a subject and a predicate. This system, that mainly encompasses first order logic (FOL), introduces the notion of universal quantifiers, and implicitly that of existential quantifiers, allows performing any deduction based on the form of the expressions.

A century later, in his article on Computational Logic [Robinson, 2000], J.A. Robinson, who had introduced the Resolution Principle in 1965, considers FOL to be "all the logic we have and all the logic we need". He points out to the central place of FOL in Logic as a discipline that had been developing since twenty centuries. He argued that "FOL can be used to set up, as first order theories, the many 'other logics` such as modal logic, higher order logic, temporal logic, …, quantum logic; and so on and so on". He considered that all these logics are based on One Logic: the first order logic; in his words "The `other logics` are simply notations reflecting syntactically sugared definitions of notions or limitations which can be formalized within FOL. There are certain universal reasoning patterns, based on the way that our minds actually work, and these are captured in FOL".

The importance that Robinson gives to FOL in the development of Logic can be extended to other disciplines such as Philosophy, in general, and the Philosophy of Mathematics in particular, precisely because Frege's main objective and so-called *programme* was to show that the whole of arithmetic can be deduced within his logical system that includes all the deductive inferences in mathematical use. In fact, Frege had a major target of reaching certainty and eliminating any conclusion uniquely based on intuition.

The direct consequence of Frege's work was the creation of a syntactic system aiming at reducing, if not eliminating, deductive errors in that system. Based on all these ideas, logical inferences could consequently be done by a pure mechanical process. As is well-known his programme of deriving the whole of arithmetic within his system turned out to be inconsistent; however, its influences on the development of computer science, many decades later, were of immense importance and remains relevant 140 years after its first publication.

The term FOL, used equivalently in this paper with Predicate Logic PL, or First Order Predicate Calculus FOPC in some other resources, has provided an artificial language with precise grammar rules that could be considered "the ancestor of all programming languages in use today" as Martin Davis said in his book `The Universal Computer` [Davis, 2000].

The notions of predicate logic, formalization, syntax, deductive reasoning and inference rules, made possible the idea of developing mechanical computational processes. These processes are simply the raison d'etre in computing science. Consequently, FOL has been impacting a wide number of subfields in computer science. As we have mentioned in the abstract, the Logic Programming Languages, PROLOG, and the Automated Theorem Proving Systems will be brought into prominence in this paper.

After more than two decades of discoveries and development in computer science, J.A. Robinson defined in 1965 [Robinson, 1965] the clausal form, a special form of predicate logic, with a single rule of inference. Robinson's *improved* resolution principle is a cornerstone in logic programming languages; it has initiated a decisive consequence in the development of PROLOG (Programming in Logic) by Colmerauer and Roussel [Colmerauer, 1972]. These developments including Kowalski's interpretation [Kowalski, 1974] represented a decisive tool in implementing machine learning programs based on Muggleton's notion of inductive logic programming [Muggleton, 1992]. In addition, the resolution principle turned out to be a very effective tool of Automated Theorem Proving, a sort of effective logic programs used to verify and prove important, usually hard, constraints. Most of these computer notions are the consequences of Frege's predicate logic.

In the first part of this paper, we introduce Frege's predicate Logic (often denoted by FOL) using its modern syntax and showing the high expressiveness it can have as opposite to what had been used before its introduction. In the second part, we expose Kowalski's interpretation of the predicate logic as a programming language, followed by putting some notions, such as the Horn clauses and the resolution principle, at the center of PROLOG and the ATP developments. These two topics are respectively discussed in detail in the third and the fourth parts of the paper. The conclusion highlights some future investigations on the role of FOL in other subfields of computer science.

2. Overview of Frege's predicate logic

his section introduces the main terms used in the predicate logic and shows its high level of expressiveness by giving some examples written in modern predicate logic that might not have been sufficiently precisely expressed before Frege's development of his logic.

A *term* can be a simple name of object ('5'), a *complex term* representing an object ('5-2'), or a sentence which is a well formed sequence of complex terms. These complex terms are usually considered as *functions* having *arguments*. Any concept is an *n-place* function which maps every argument to a *truth-value*, true or false, and produces a truth-value.

In Frege's predicate logic, a predicate is analyzed as *concept* which is a special case of a function. In this sense, Frege's logic is a functional application rather a predicate application which is exactly the opposite of the modern use of functions and predicates (or relations).

In addition, more complex expressions requiring for example the negation and the conditional are possible in the Frege's notations. Frege used the identity sign to represent the equivalence of two conditions. This is usually replaced by the biconditional in the modern texts.

On the other hand, the *quantification* is fundamental in Frege's logic. The use of 'every' and 'some' makes his logic deal with more complex inference rules. Note that Frege has not introduced the existential quantifier in his symbolic system, but this is obviously obtained by using the negation and the universal quantifier.

It is important to mention that in Frege's logic a relation Rof the form $\exists x, R\ (x, instance)$ or of the form $\exists x, R\ (instance, x)$ are both instances of a single inference rule. Formally, this relation can capture these two inferences as follows: $R\ (a_1, a_2, ..., a_n) \rightarrow \exists x, R\ (a_1, ..., x, ..., a_n)$ where the relation R has possibly n arguments, and $a_1, a_2, ..., a_n$ are constants $\forall a_i, 1 \leq i \leq n$.

This means that the variable x can precede or follow the constant *instance* in the list of arguments of R without modifying the structure of the inference rule. The variable x is *bound* (belonging to a values domain) by what is called variable-binding operators "some x is such that".

In addition, Frege's logic permits to express ideas like "every function f is such that" and "some function f is such that". This possibility, that allows quantification over functions similarly over objects, describes, in fact, a second-order predicate calculus. For example, if the two arguments x and y fall under the concept function f, then this second-order predicate calculus expression can be written as $\forall f, f(x) \leftrightarrow f(y)$.

Furthermore, the use of nested quantifiers in Frege's logic is one of the most remarkable new possibilities. With Frege, the quantifiers can be combined to express more complex propositions. For example, the difference between the two sentences "Every man loves a woman" and "There is some woman whom every man loves" is easily expressed without any

ambiguity by using two nested quantifiers which was not possible to express accurately before Frege. These two sentences are, respectively, expressed as follows using modern symbols:

$$(\forall x)(\exists y)\ (Man(x) \rightarrow (Woman(y) \wedge Loves(x,y)))$$
$$(\exists y)(\forall x)\ (Woman(y) \wedge (Man(x) \rightarrow Loves(x,y))$$

Based on this powerful expressiveness of the FOL, many definitions in Mathematics became very precise using nested quantifiers. A typical example in calculus is the difference between the definitions of the continuity and the uniform continuity of a function. Using the modern symbols of Frege's predicate logic, these two definitions can be accurately expressed using nested quantifiers as follows:

The continuity of *f*: $(\forall\varepsilon)(\forall x)(\exists\delta)(\forall y)(|x\text{-}y|<\delta\rightarrow|f(x)\text{-}f(y)|<\varepsilon)$

The uniform continuity of *f*: $(\forall\varepsilon)(\exists\delta)(\forall x)(\forall y)(|x\text{-}y|<\delta\rightarrow|f(x)\text{-}f(y)|<\varepsilon)$

It is important to mention two fundamental properties in Frege's predicate logic: the *soundness* and the *completeness*. A deductive system is called complete if every logically valid formula is the conclusion of some formal deduction. In other words, if the output is satisfiable, then the inputs must also have been *satisfiable* (if it is possible to assign an interpretation making the formula true).

Conversely, the soundness is the fact that only logically valid formulas are provable in the deductive system. This means if the inputs of an inference rule were satisfiable, then its output would also be satisfiable.

Fortunately, the predicate calculus is *sound* and *refutation complete* (the predicate logic system is able to derive false from every set of formulas which are *unsatisfiable)*. This means that given a formula in predicate logic with no interpretation that can make it true, it follows that the system *can* derive *false* from that formula. The idea of refutation is central in automated theorem proving and will be discussed below in the text. These two fundamental properties of the predicate logic are primary due to the Herbrand and Gödel theorems.

Hence, based on these fundamental definitions, a syntactic and semantic correspondence between the predicate logic and programming languages can be shown now. This correspondence is known to be the Kowalski's interpretation and is elaborated upon in the following section.

3. Predicate Logic as Programming Languages

It is fascinating to see how a formal first order logic system developed by Frege for a philosophical status of Mathematics is *interpreted* one century later as a computer programming language. Kowalski's interpretation considers part of the predicate logic supplemented by Robinson's principle discussed below to give, with Van Emden, syntactic and semantic meanings to the predicate logic which is *understood* as a programming language. In this section we start introducing this interpretation before discussing the Robinson's principle.

3.1 Predicate logic and Kowalski's interpretation

In his celebrated article "Predicate Logic as Programming Language" written in 1974, R. Kowalski gave an interpretation of Predicate Logic as a programming language based on the interpretation of the sentence $(A_1 \wedge A_2 \wedge ... \wedge A_n) \rightarrow B$ as procedure declaration, where B represents the procedure name, and A_1 to A_n are the procedure calls representing the body of the procedure.

In this interpretation, a problem to be solved is seen as a theorem to be proved, and a proof is a set of computations generated by a Theorem-Proving mechanism which executes the program put in the axioms.

Importantly, Kowalski's interpretation is based on the notion of *clausal form*. An expression or sentence in clausal form is a set of clauses, and a single clause is a pair of sets of atomic formulas, called also literals, having the following form:

$$A_1, A_2, ..., A_n \rightarrow B_1, B_2, ..., B_m \qquad (1)$$

In this context, an atomic formula is a *k-place* predicate symbol of k terms of the form $P\ (t_1, t_2, ..., t_k)$. Every term t_i can be a variable *or an h-place* function symbol where the function's arguments are terms. The predicate symbols, the function symbols, and the variables are mutually disjoint.

It is also important to interpret the semantic of the *set of clauses* as a conjunction $C_1 \wedge C_2 \wedge ... \wedge C_n$ and each clause of the form (1) with k variables $x_1, x_2, ..., x_k$ is interpreted as an implication using a universal quantifier. This clause can be described as $\forall x_i\ i = 1 ... k$: $A_1 \wedge A_2 \wedge ... \wedge A_n \rightarrow B_1 \vee B_2 \vee ..., \vee B_m$.

Remark that the neutral truth-value of the conditional part $A_1 \wedge A_2 \wedge ... \wedge A_n$ is true, and the neutral truth-value in the conclusion $B_1 \vee B_2 \vee ..., \vee B_m$ is False. This remark will be used below in the text when analyzing the different cases of these clausal forms.

Note that this form is equivalent to a disjunction of terms based on the fact that $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$, and a sentence of clauses, seen as conjunction of disjunctions, is usually called a CNF (Conjunctive Normal Form or Clausal Normal Form).

In the same paper, Kowalski considers that a clausal form constitutes a natural language that expresses thought in line with what Robinson has already done by introducing efficiently the resolution principle.

In the above, we have given an overview of the language's syntax, i.e. the form in which a sentence can be written, of a predicate logic sublanguage. In order to understand the meaning of these sentences using a programming language terminology, another discussion of the semantic of that sublanguage is required. This is the purpose of the next section that matches every form of a predicate logic sentence with its behaviour when executed in a logic programming language.

Such a sentence is written in so-called Horn clauses, previously pointed to by A. Horn in 1951, turned out to be fundamental in Logic programming languages (see infra). These special sorts of clauses are discussed in the following subsection.

3.2 Horn clauses and programming semantics of Frege's logic

In this paragraph, we analyse all the possibilities of a clausal form written in a suitable form, the Horn clause, which used in computer programming. This will prove helpful in understanding the functionalities of PROLOG language introduced in section 4.

A Horn clause is a clause with at most one positive literal in the conclusion (or equivalently at most one positive literal when the clause is written as a disjunction of terms. This definition, describing a sub-language of predicate logic, generates four different cases of a Horn clause.

The semantic interpretation of these cases in term of programming language meanings represents two ideas. The first is a basis theoretical *correspondence* of a subset of predicate

calculus form, written as a Horn clause, and programming language. The second is a fundamental contribution to the development of a Logic Programming Language, PROLOG.

This point will be re-discussed below in the text.

The four cases of a Horn clause are the following:

1. The *null clause* where no literals are used in the condition neither in the conclusion. In programming language interpretation, it corresponds to a halting instruction or reaching-the-goal instruction. In a procedural meaning, it is an anonymous procedure without body.
2. No literals in the condition and one literal in the conclusion. This takes the form $\boxed{\rightarrow B}$, which is equivalent to $\boxed{true \rightarrow B}$ and also equivalent to $\boxed{\neg true \vee B}$ corresponding to exactly one positive literal (B). Again remember that True has replaced the empty part as a neutral truth-value among the conjunctions. The meaning of this form is simply a Fact. It corresponds to a procedure without body.
3. When there is n literals in the condition and no literals in the conclusion. The sentence form is $(A_1 \wedge A_2 \wedge ... \wedge A_n) \rightarrow$; this is equivalent to $(A_1 \wedge A\_2 \wedge ... \wedge A\_n) \rightarrow false$. Its interpretation is seen as a goal instruction. The goal consists of executing the list of procedure calls A_1, A_2, ..., A_n. In this case, the main procedure is considered anonymous.
4. The more general case is $(A_1, A_2, ..., A_n) \rightarrow B$. This case with exactly one positive literal (B) is usually called *definite clause.* As used before, the conclusion B corresponds to a procedure name, and the condition $(A_1, A_2, ..., A_n)$, the procedure body, is a set of n calls.

This discussion leads us now to introduce the resolution principle, a powerful way to generate a solution from a set of clausal sentences by, roughly speaking, finding two identical predicates and trying to substitute a variable by a constant. We will see that these two operations, Matching/Instantiating, constitute the computational core of PROLOG.

3.3 The role of the resolution principle in logic of programming languages

As previously mentioned the importance of the resolution principle in logic programming language stems from the fact that it has been developed as specific operations applied on a subset of the predicate logic. Its main objective is to provide a powerful mechanical process in a computerized theorem-proving. This principle has been improved by Robinson and played a central role in Kowalski's work discussed supra. Even though Robinson's work precedes that of Kowalski, the high impact of this principle on PROLOG language led us to introduce these concepts in this order, on our way towards section 4 that will focus on PROLOG functioning in some detail.

The Resolution Principle (RP) is an inference rule used in theorem proving by refutation (i.e. tries to find a contradiction of the negated rule to be proven). This principle deals with Skolemized formulas of the form $\forall x_1, ..., \forall x_n, X$, where X is a CNF formula without quantifiers.

In propositional logic, Modus Ponens can be considered as a special case of the resolution principle: $((p \rightarrow q) \wedge p) \rightarrow q$ or again $((\neg p \vee q) \wedge p) \rightarrow q$. The general case is a single inference rule that entails a new clause from two clauses containing complementary literals $(p \wedge \neg p)$.

Interestingly, a generalized approach can be applied to clauses in predicate logic. The resolution principle in predicate logic is based on a key technique called the *Unification*. The Unification is a sort of matching operation that takes two terms and tries to make them identical by instantiating the variables in both terms.

For example, if we have $\forall x, (A(x) \rightarrow B(x)) \wedge A(t)$ we can obtain $B(t)$ when applying the resolution principle and using the unifier $[x/t]B(t)$.

Remark that usually this is written by omitting the quantifiers and by using the CNF form. It becomes $((\neg A(x) \vee B(x)) \wedge A(t)) \rightarrow B(t)$.

This Unification allows the creation of *factors* which reduce the duplication of terms.

For example, $P(x) \vee \neg(Q(f(x), a)) \vee P(g(y))$ is factored to $P(g(y)) \vee \neg(Q(f(g(y)), a)$.

In terms of Kowalski's interpretation, the resolution principle is seen as procedural semantics.

The procedure invocation can be used to generate new procedure (or assertions). For example, given a procedure A_i that matches a procedure name B using a substitution function f with $(A_1, A_2, ..., A_n) \rightarrow A$ and $(B_1,B_2,...,B_m) \rightarrow B$, a new procedure is generated as follows: $(A_1, A_2, ..., A_{(i-1)}, B_1,B_2,...,B_m, A_{(i+1)}, ..., A_n)f$ where f is a function that substitutes a variable by a term.

The Horn clauses form and the resolution principle are of high importance in computational logic in general, and in the automated theorem proving in particular. One reason of the effectiveness of the Horn clauses in theorem proving is the fact that the resolvent, the produced clause by resolution, of two Horn clauses is itself a Horn clause, and the resolvent of a goal clause and a definite clause is a goal clause.

While it is known that the resolution principle was already discussed by Davis and Putnam in 1960, their algorithm worked unfortunately in an exhaustive instantiation way over a formula which creates what is called a "combinatorial explosion". Robinson's effective solution used a unification algorithm that allows the instantiation process to act during the proof until the finding of a contradiction. By doing that, this "combinatorial explosion" has been avoided.

In sum, the refined resolution principle, based primary on Frege's first order logic, has perfectly prepared the theoretical foundations to create a new logic programming language. This language is called PROLOG and remains used today in Artificial intelligence and many other applications.

4. The place of Frege's logic in PROLOG'S origin

PROLOG (Programming in Logic), a programming language based on the predicate logic is perhaps the most natural and representative example of the influences and the "success" of Frege's logic in computer science. It was developed in 1972 by A. Colmerauer with P. Roussel with a first intention of creating a declarative language. The program is basically expressed as facts and rules to describe relations, and a query that generates a sequence of computations over these facts and rules. It is a sort of formulas assumed true and representing the premises in predicate logic. These formulas are written as Horn clauses. When trying to answer a query, PROLOG builds a search tree using the resolution and unification principle. The procedural semantics of PROLOG is a procedure trying to satisfy a set of goals. The procedure lists the truth or falsity of this set of goals and their respective unifications (or instantiations). A very important mechanism in PROLOG is the automatic backtracking in order to find other solutions. This idea is, by itself, a corn stone in the development of predicate logic evolving from a formal system describing truths to a sort of mechanical process to generate new truths from a set of premises and inference rules.

In PROLOG, a definite clause works as a goal-reduction procedure, and its syntax is shown in reverse form as exactly used by Kowalski. For example, the definite clause $B \leftarrow (A_1 \wedge A_2 \wedge ... \wedge A_n)$ is interpreted as "to prove B, prove $(A_1 \wedge A_2 \wedge ... \wedge A_n)$" to highlight

the fact that to prove the goal B, the literals $(A_1 \wedge A_2 \wedge ... \wedge A_n)$ have to be proved before. This corresponds again to Kowalski's semantics interpretation discussed above when B represents the procedure name and A_1 to A_n are the procedure calls. This Horn clause is represented in PROLOG as follows:

```
B: - A1, A2, ..., An.
```

On the other hand, the proof by refutation is a central idea in PROLOG processing. For example, to prove a goal clause C in a program P, PROLOG adds the negation of C $(\neg C)$ to the program and tries to find a contradiction (represented by false as result). So in order to prove C, PROLOG refutes not C. In the very used problem of solving the existential quantifier over a list of positive literals $(\exists x(a \wedge b \wedge ... \wedge k))$, the negation of this sentence will be treated as $(\forall x(\neg a \vee \neg b \vee ... \vee \neg k))$ or again $(\forall x(\text{false} \leftarrow (a \wedge b \wedge ... \wedge k)))$ and represented as ":- a, b, …, k".

Note once again that this existential clause (written here in Horn clause) was also replaced in the original Frege's system by the universal quantifier over the negation of the clause.

In this context, it is pertinent to mention that the resolution-based theorem proving uses the proof by refutation approach. In order to prove a sentence, it can be more suitable to obtain a contradiction when its negation is added to the system. This is often the case in computational logic. For that reason, most automated theorem proving use the proof by refutation.

The following section algorithmically highlights how PROLOG performs its computation when receiving a set of premises and how it applies the resolution principle.

5. Predicate logic and computational aspects of the resolution principle

In this section, in addition to illustrating the resolution principle working in PROLOG through a detailed example, we develop a reflection on the comparison of the human-oriented and the machine-oriented reasoning in a deductive system. We will see, for example, why the predicate logic using the resolution principle is very effective when executed by a computer. In this sense, the computational aspect and relevance of applying such a principle on a first order logic is highlighted.

As we have seen, the Horn clauses, a subset language of predicate logic combined with the principle of resolution/unification constitute the heart of computation process in logic programming languages such as PROLOG.

The mechanical proof of a formula or the Resolution proof algorithm can be described as follows:

1. Given a set of theorems as a knowledge base (KB) written in conjunctive normal form (CNF), and a formula F to be proved, the negation of F is added to the KB.
2. Repeat until no more clauses to be resolved:
    - Find resolvable clauses by applying the resolution/unification principle (matching/instantiation) and add the result (the resolvant) to the KB.
    - If the resolvant is false (NIL), declare the formula F true (because its negation is false)
3. Declare the formula F false.

Note that in the preceding algorithm a decision of type True/False is given without explicitly listing the many possible solutions that a potential conjecture can generate. This is exactly the case in the following example in which we go a little further than finding a simple Yes/No answer. We introduce a KB having a main request conjecture with a variable. This means that the procedure's objective is not only to check the validity of the conjecture, but also to generate all possible solutions (if they exist) by finding the list of values that can be instantiated to this variable. In particular, the sentence Loves (Marc, x) initiates a search process of all "values" of x that Marc loves.

Who does Marc love?

∃ x: Loves (Marc, x)

The Knowledge base written in clausal form is as follows:

1. ¬Sees (Marc, v) ∨ Loves (Marc, v)
2. Sees (Marc, Mary)
3. Sees (y, Alice)
4. Sees (z, Mother(z))

The given #5 is the negation of the original conjecture to be proved. Using a variable x it is seen as a question that calculates all possible values of x.

5. ¬Loves (Marc, x) (since ¬ ∃x: Loves (Marc, x) <-> ∀ x: ¬Loves(Marc,x))

The resolution with 5 and 1 unify(Loves(Marc, x), Loves(Marc, v)) = {x/v} means x is substituted by v and gives:

6. ¬Sees (Marc, v)

Three possible resolutions can be calculated:

- With 6 and 2 unify(Sees(Marc, v), Sees(Marc, Mary))= {v/Mary}
- With 6 and 3 unify(Sees (Marc, v), Sees (y, Alice)) = {y/Marc, v/Alice}
- With 6 and 4 unify(Sees (Marc, v), Sees (z, Mother(z))) = {z/Marc, v/Mother(z)}

These 3 unifications give, respectively, three possible solutions:

- Loves(Marc,x) with {x/v, v/Mary} (i.e. Marc loves Mary)
- Loves(Marc,x) with {x/v, y/Marc, v/Alice} (i.e. Marc loves Alice)
- Loves(Marc,x) with {x/v, z/Marc, v/Mother(z), } (i.e. Marc loves his mother)

This example shows at least two significant facts. The first of these is that it only uses the resolution principle based on Unifying/Instantiating rules written in clausal forms. Secondly, if these rules become much more complex, the mechanical deduction process remains exactly the same. These two points, complex rule/one mechanism, constitute a very suitable Machine-oriented approach in order to obtain new deductions. This approach, when implemented with specific search algorithm such as a backtracking procedure, can be executed in a very effective way.

Typically, a Human-oriented deduction is based on a sequence of simple deductive steps each of which is easy to be "humanly" verified based on the premises, the preceding steps, and also simple rules of inference. The number of steps generated to obtain a final deduction could be very long, but it is still possible to use by an intuitive-human mechanism.

However, this mechanism, due to various reasons, is absolutely ineffective when executed by a computer. The first of these is that computers, for instance!, have no "intuition". Secondly, a potentially big number of steps based on different rules of inference easily creates a *combinatorial explosion* in term of execution effectiveness. Additionally, a more complex but few number of inference rules are much better executed by a computer. This is exactly how and what a resolution principle produces when implemented in a modern computer. The combinatorial explosion is, then, avoided and more effective deductions are obtained. The resolution principle has also been already refined by Kowalski and Kuehner to obtain linear deductions (SL-Resolution) showing a very high performance in practical implementation. An example of implementing the linear resolution is the Edinburgh Structure-Sharing Linear Resolution Theorem Prover.

To give the reader a sense on the widespread disciplinary use of FOL, the following last part discusses one of the most successful applications of FOL in computer science, the automated theorem proving. Those powerful computer programs use, generally, many concepts discussed previously, in particular the proof by refutation combined with the resolution principle.

6. Predicate logic and automated theorem proving systems

Another significant consequence of the predicate logic development in computer science is the automated theorem proving systems.

In this section, we describe these systems by linking them mainly to the first order logic, and then we discuss the use of such programs to solve theoretical problems as well as solving industrial problems. We continue by giving some known ATP systems and representative applications where these systems are mostly used.

6.1 Description of automated theorem proving

Automated Theorem Proving (ATP) aims at the implementation of software that proves a conjecture assumed to be a true statement, to be logically derived from a set of premises. If this proof succeeds, the conjecture is then called theorem. The most used logical language in which the conjecture and the premises are introduced is a first order logic, but it could be written in higher order logic or a different logic's type.

In line with the preceding discussion showing the precision and the expressiveness of the first order logic, the formal way of accurately expressing the required statements (conjecture and premises) is a central strength of ATP. The problem is submitted without any ambiguity, and the proof mechanism of ATP tries to find a solution by applying exactly some of the computation approaches discussed above in this paper such as CNF, Resolution and Unification principle, etc.

In most of ATP systems, the proof of a theorem or the disproof of the conjecture is followed by a detailed description of the steps that lead to the final conclusion. The user, when following these, usually very large, steps, can better understand why the conclusion is derived or not from the premises. Moreover, sometimes these detailed steps constitute by themselves a more interesting solution to the problem than a simple Yes/No answer. For example, given an initial configuration of a game with an objective to check if another configuration can be reached (i.e. a winning configuration), the provided steps by the ATP proof constitute exactly the steps to follow in order to solve the problem (to win in this example). These points make, generally the ATP systems very powerful computer programs able to solve hard problems using an "acceptable" time complexity. The ATP techniques are diverse. The most popular

techniques are, among others, First-order resolution with unification, Model elimination (Loveland), Model checking, Higher-order unification.

6.2 Main applications of ATP systems

The ATP systems or what is called *Provers* are used in different domains, and the main successful uses were related to logic, mathematics, computer science, and engineering. Originally, the ATP systems have been implemented to prove mathematical conjectures. Nowadays, it covers a very large variety of domains, but the most used applications concern certainly formal methods for software verification and design, and hardware verification. The use of ATP in this type of application is of huge commercial importance. It could prevent costly many errors that potentially exist in, for example, a hardware integrated circuit implemented in any scientific machine's component (medical instrument, robot, nuclear controller, space engine, etc.).

Many applied successful examples of ATP systems in mathematics, software and computer engineering are given below in the text.

6.3 Propositional provers versus first-order provers

The difference of expressiveness between the propositional logic and the first order logic remains significant when these two logics types are used to implement two different provers of the corresponding logics. This section discusses these two types of provers without ignoring some basic technical improvements that first order provers have realized.

A propositional prover (or SAT solver), based in propositional logic, is basically a Boolean satisfiability solver where the problem is expressed as Boolean terms. The SAT solvers (such as zChaff) are very useful but the Boolean expressions become rapidly very large and the expressiveness of such expressions is generally very limited. These two points constitute the basic weaknesses of a typical SAT solver. Note that the SAT problem turned out to be the first proved NP-complete problem, which generally takes an exponential complexity to solve a substantial problem.

On the other hand, a First order logic theorem prover, which includes quantifiers and predicates, is much more expressive than a SAT prover. The main strength is the Robinson's resolution principle implemented in first order logic that makes the steps of a search proof mechanical. This category of provers is certainly one of the most mature and developed parts of the ATP systems. A problem is usually expressed in a natural way using an expressive specification.

It is useful to mention that some problems requiring equality for better expressiveness are not always running effectively under a classic first order logic prover. Consequently, many modern provers including equality (Equational theorem provers, such as E prover) have been developed to overcome this obstacle. In general, these sorts of provers implement specific search heuristics for a better performance. This approach that uses search heuristic and based on first order logic prover is very applied in modern machine learning, an ability given to computers that act and learn without being explicitly programmed.

Importantly, one of the most significant and first successes of first order logic provers was software verification. Specifically, the provers verify the correctness of programs written in different programming languages such as Ada, Pascal, or Java. One well-known example of program verification, Stanford Pascal Verifier, was implemented by David Luckham at Stanford University and based similarly on Robinson resolution principle.

Let us now select some renowned provers and discuss their characteristic and successes. The following section is dedicated for this objective.

6.4 Some well-known provers in historical prospects

In this paragraph, we discuss some basic examples FOL provers. One of the oldest provers (launched in 1989) is SETHEO, which was developed at the Technical University of Munich. It is a high-performance system based on the goal-directed model elimination calculus.

Another FOL ATP is Vampire, which was developed in 1994 by Voronkov-Hoder and formerly by Riazanov at Manchester University. Vampire is used to solve a large variety of problems, but it is well effective in software verification, hardware design and verification, knowledge representation and reasoning, and Mathematical theorem proving.

The E prover is yet another example of a high-performance prover and friendly reasoning system. It was developed in 1998 by S. Shulz at Technical University of Munich. It is a full first-order logic prover, but built on a purely equational calculus.

EQP is another first order and equational logic prover designed at Argonne National Laboratory and very known to have proved Robbins' conjecture in 1998. One of its characteristics is the effectiveness of performing associative-commutative unification and matching operations.

As for the worth-mentioning Waldmeister prover is a unit-equational first-order logic developed at Max Planck Institute for Computer Science in 1998. It is known to have a high performance in terms of time and space complexity.

Another interesting example is that of W. McCune, at Argonne National Laboratory, who implemented Prover9 in 2004 as a successor of Otter and first-order and equational logic ATP. This prover has later been paired with Mace4.

Finally, SPASS, first order logic theorem prover with equality, has been launched in 2010 by Research Group Automation of Logic at Max Planck Institute for Computer Science, was extended to SPASS-XDB to incorporate facts from relational databases.

On the other hand, several higher order logic provers have been developed and accomplished, during the last 50 years, enormous number of tasks in, among others, Mathematics and engineering. ACL2, Coq, HOL, and Nqthm represent some of these provers.

As we have just seen, the use of provers based on Frege's first order is widespread and represents an essential tool in computer science. The ATP systems are used in a very large number of applications. The next subsection shows different domains that successfully use the ATP systems.

6.5 ATP systems: some successful examples

Historically, ATP systems have been solicited to prove interesting and hard problems in Mathematics (Graph theory, Group theory, Algebra, Geometry, etc.), Software design and verification, hardware verification, etc. The following selected examples show the wide range of applications in which ATP systems, and particularly FOL provers, are increasingly used to prove harder and costly important problems.

In the following, we list some successful ATP systems in various disciplines and fields.

Mathematics

- The FOL prover Otter has proved many results in quasi-groups and used by K. Kunen.

- The FOL prover Otter has also been used by McCune to find minimal axiom sets and to solve other problems in Algebraic structures.
- The Japanese system ICOT helped to decide many quasi-groups problems and used by Fujuta-Slaney-Benett.
- The EQP (the equational Prover), a FOL prover, has been used to finally solve the Robbin's problem conjecturing that a specific set of axioms constitute a basis for Boolean Algebra.
- Geometry Expert, a geometry prover, has proved new results in Euclidean geometry.
- NUPRL, a higher order prover, has proved Higman's lemma and Gerard's paradox.

Software Design

- The project *Amphion*, financed by NASA, using Knowledge-Based Software Engineering, and based on formal methods (roughly provers) in order to automate software reuse that improve software productivity and quality.
- The system KIDS used to create scheduling algorithms using operations permitting the development of code from the specifications. This system was designed at Kestrel Institute.

Software verification

- The PVS system deals with scheduling applications and specification of space flight control procedures.
- The *KIV* (Karlsruhe Interactive Verifier) is used for different software verification applied to special functions, graph manipulation, verification of PROLOG program under WAM (Warren Abstract Machine).
- The *Key* project deals with the verification process related to object-oriented paradigm.

Hardware verification

- Bell laboratories used the higher order logic HOL system as interactive environment theorem proving for industrial hardware verification.
- Some proofs of the correctness of codes related to the floating point division have been tested using the ACL2 system. These codes concerned specific types of AMD microprocessor. Similar type of verification has been done by Nqthm for the correctness of the FM9001 microprocessor.

This section has shown some representative fields in which the ATP systems are successfully used mainly by the experts for a research interest or for an industrial purpose. However, some ATP systems are now accessible online for different categories of users. The TPTP (Thousands of Problems for Theorem Provers) is an important example of an online library interface to many ATP systems developed at the University of Miami.

Conclusion

This paper has mainly shown the indispensable role Frege's predicate logic has played in the foundations and development of the most popular logic programming language, PROLOG.

The predicate logic supplemented by logic programming languages has led to the development of one of the most significant successful FOL-based fields in computer science: the ATP systems. The list of computer science applications that have benefited from the predicate logic is very long. Probably, each of these applications deserves a dedicated investigation.

For example, the Machine Learning in artificial intelligence, a capacity of computer systems to learn and act without preceding exhaustive programming took originally its basis from the FOL and logic programming language. In particular, the works of Muggleton in the early 1990s led to develop a special form of Machine Learning, inductive supervised machine learning called Inductive Logic Programming (ILP) that has logic programs as inputs and outputs.

A fascinating example of ILP that also deserves more investigation is program induction. Its main objective becomes not only to deal with examples but to induce a program from many examples of inputs and outputs.

These modern applications of computer science that took their basis from Frege's predicated logic and improved by other concepts prove, once again, that this logic type remains today relevant and fundamental in several computerized models.

🕮 References

Colmerauer, Alain; Kanoui, Henri; Passero, Robert, and Roussel, Philippe. *Un système de communication homme-machine en français* (french human-machine communication system), AI research group, Aix-Marseille university, Luminy, 1972.

Curry, Haskell, and Feys, Robert. *Combinatory Logic.* North-Holland, 1958.

Curry, Haskell; Hindley Roger, and Seldin Jonathan. *Combinatory Logic*, Volume 2. North-Holland, 1972.

Davis, Martin. *Influences of mathematical logic on computer science*, In R. Herken, editor, The Universal Turing Machine: A Half-Century Survey, pages 315–326. Oxford University Press, 1988.

Davis, Martin. *The Universal Computer*. Norton, 2000.

Davis, Martin. *Engines of logic: mathematicians and the origin of the computer.* New York: W.W. Norton & Co., 2001.

De Mol, Liesbeth, and Primiero, Giuseppe. *Facing computing as technique: towards a history and philosophy of computing*, Philosophy & Technology, 27, 3, pp 321-326, 2014.

De Mol, Liesbeth, and Primiero, Giuseppe. *When logic meets engineering: introduction to logical issues in the history and philosophy of computer science.* History and Philosophy of Logic, 36 (3): 195-204, 2015.

Frege, Gottlob (1879). *Begriffsschrift*, Eine der arithmetischen nachgebildete Formelsprache des reinen Denkens, English translation in Jean van Heijenoort (ed.), From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931, Harvard University Press, pp. 1-82, 1977.

Gillies, Donald. *Logicism and the development of computer science*. A. C. Kakas and F. Sadri (eds,): Computational Logic, Springer-verlag Berlin Heidelberg, pp. 588-604, 2002

Knuth, Donald. *Computer science and its relation to mathematics*. American Mathematical Monthly, 81, 323-343, 1974.

Kowalski, Robert. *Predicate logic as programming language*, Proc. IFIP Cong 1974, North-Holland Pub Co, Amsterdam, pp 569-574, 1974.

Mancosu, Paolo; Zach, Richard, and Badesa, Colixto. *The development of mathematical logic from Russell to Tarski*, 1900-1935, in: Haaparanta, L., ed., The development of modern logic. New York: Oxford University Press, pp. 318-470, 2009. Available online at: www.ucalgary.ca/rzach/static/history.pdf.

Manna Zohar, and Waldinger Richard, The Logical Basis for Computer Programming. Addison-Wesley, 1985.

Muggleton, Stephen (ed.), Inductive Logic Programming. Academic Press, 1992

Robinson, John Alan, A Machine-Oriented Logic Based on the Resolution Principle. Journal for the Association for Computing Machinery, 12, pp. 23-41, 1965.

Robinson, John Alan. *Computational Logic: Memories of the past and challenges for the future*. J.Lloyd et al. (Eds): CL 2000, Springer-Verlag Berlin Heidelberg, pp.1-24, 2000.

Tedre, Matti. *The Science of Computing Shaping a discipline*. Boca Rato: CRC Press, 2015.

Ulam, Stanislav. *Von Neumann: the interaction of mathematics and computing in:* J. Howlett, N. Metropolis and G.-C. Rota, eds., A History of computing in the twentieth century 1980, Proceeding of the international research conference on the history of computing, Los Alamos, New York: Academia Press, pp. 93-99, 1976