Inference Using Formal Logics

David McAllester

March 21, 2001

1 Introduction

Logic is fundamental to a variety of disciplines. Logic provides insight into the nature of mathematics and human mathematical reasoning. Logic provides insight into the syntax and semantics of human language. Logic is an important tool in the design and implementation of computer programming languages. Logic also holds the promise of playing an important role in the development of artificial intelligence. I will briefly consider each of these motivations in more detail.

Metamathematics. The idea that human thought is governed by linguistic rules starts with the development of syllogisms by Aristotle. A syllogism is a pattern of inference. For example, if all men are mortal, and Socrates is a man, then Socrates is mortal. The general pattern is, of course, if all Xs are Ys and Z is an X, then Z is a Y. In modern terminology, such a pattern is called a *rule of inference*. The study of inference rules, began by Aristotle, lead to the development in the 1920's to the system of rules and axioms known as ZFC — Zermelo Fraenkel set theory with the axiom of Choice. It is now generally accepted by mathematicians that all the provable truths of mathematics can be formally derived from the nine axioms of ZFC using the inference rules of first order logic. Gödel's incompleteness theorem, proved in 1930, implies that if ZFC is sound then there are true statements not provable in ZFC. Gödel's theorem is completely consistent with the idea that the theorems provable by human mathematicians are precisely those provable in ZFC. True statements unprovable in ZFC and presumably also unprovable by human mathematicians. Gödel's theorem is itself provable in ZFC. A general treatment of logic as metamathematics can be found in [2].

Metaprogramming. A compiler is a computer program that takes as input another computer program and compiles that input into low level instructions executable by computer hardware. A compiler is a metaprogram — a program that manipulates and analyzes other programs. Logic plays an important role in metaprogramming. Computer programs are composed of symbolic expressions which have well defined meanings and where the meaning of an expression is determined by the meaning of its subexpressions. Furthermore, compilers often analyze (reason about) the programs being compiled. For example, many compilers can determine that a pointer variable must be non-null (not equal to zero) at a certain place in the program. Logic is fundamental to such software analysis. A survey of the theory and practice of compilers can be found in [1].

Human Language. All human languages have sentences which form "complete thoughts". Furthermore, in all languages sentences have phrase structure, e.g., a sentence may be composed of a noun phrase and a verb phrase each of which in turn can be composed of subphrases. It seems that the meaning of a sentence is constructed from the meanings of its phrases and the meaning of a phrase is constructed from the meaning of its subphrases. The phrase "book on the table" has a meaning that is determined by the meaning of the word "book" and the meaning of the phrase "on the table". The ability to combine meanings by putting words together into phrases, and phrases together into larger phrases, allows for a combinatoric number of different ideas to be expressed — the number of ideas expressible in a fixed number of words grows exponentially in the number of words allowed. Logic provides a clear (though approximate) mathematical model of the way meanings are assigned to phrases and the way the meaning of a phrase is determined by the meanings of its subphrases. An in-depth study of the logical treatment of human language can be found in [3].

Knowledge Representation. Since the early days of artificial intelligence logic has been put forward as a foundation for the representation of knowledge about the world [7]. Despite some early hopes, logic-based mechanical deduction has not produced human-level intelligence in machines. But neural nets, genetic algorithms, and reinforcement learning have also failed to produce human-level intelligence, e.g., machines that that can have sustained intelligent conversations in human language. Because of the fundamental importance of logic to metamathematics, metaprogramming, human language and knowledge representation, logic remains relevant to the study of intelligence.

2 What is logic?

Logic is the study of the manipulation of *symbolic expressions*. A symbolic expression is a concise representation of a much larger (often infinite) set of behaviors or responses. For example, the symbolic expression $x^2 + 2x + 1$, i.e., the text whose first character is the letter "x" and so on, is a concise representation of an infinite input-output relation — the relation mapping input x to output $x^2 + 2x + 1$. Algebraic rules of inference allow one to deduce that if $2y + 4 = 4x + 6$ then $y = 2x + 1$. The symbolic expression $2y + 4 = 4x + 6$ is a proposition — an expression that, given values for its variables, is either true or false. To say that a proposition Φ *entails* a proposition Ψ means that in any situation where Φ is true, we have that Ψ must also be true. For example $2y + 4 = 4x + 6$ entails $y = 2x + 1$.

Formally, a symbolic expression can be defined as a tree where each node is labeled by a character string. For example, the expression $2y + 4 = 4x + 6$ is a tree whose root is labeled with "=" and where the root has two children — one representing $2y+4$ and one representing $4x+6$. An expression whose root is labeled with the string f and with children trees $t_1, \ldots, t_n$ is often written as $f(t_1, \ldots, t_n)$. For example, the expression $2y + 4 = 4x + 6$ can be written as follows.

$$= (+(*(2,\ y),\ 4),\ +(*(4,\ x),\ 6))$$

The leaves of this tree are labeled with the numerals "6", "4", and "2" or the variables "x" or "y". The internal nodes are labeled with "=", "+" or "*".

An expression is usually used to represent a function whose inputs are the variables occurring in the expression and whose output is the value of the expression. There are many different ways of assigning values to expressions. Different ways of evaluating expressions correspond to different logics. There are logics of time, modal logics, logics of knowledge, logics of communicating systems, logics of imperative computer programs, and many more. Historically, however, the best known logic is first order logic.

3 First Order Logic

In first order logic one assumes a set of variables, a set of constants, a set of function symbols and a set of predicate symbols where each function and predicate symbol is associated with a specified arity (number of arguments). In first order logic there are two types of expressions — terms and formulas. The terms t and formulas Φ can be defined by the following grammar.

$$t := x \mid c \mid f(t_1,\ \ldots,\ t_n)$$

$$\Phi := P(t_1,\ \ldots,\ t_n) \mid t_1 = t_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \rightarrow \Phi_2 \mid \neg\Phi \mid \forall x \Phi \mid \exists x \Phi$$

In the grammar for terms we have that x ranges over variables, c ranges over constants, and f ranges over function symbols. We assume some unspecified way of classifying character strings into variables, constants and function symbols with specified arities. For example, if f is a function symbol of two arguments, g is a function symbol of one argument, x and y are variables, and b and c are constants, then $f(b,\ f(g(x),\ f(c,\ y)))$ is a term. In the grammar for formulas, P ranges over predicate symbols.

The defining feature of most logics is the *semantics* assigned to the expressions of that logic, i.e., the way in which values are assigned to expressions. In the case of first order logic the semantics is defined relative to a *structure*. A structure $\mathcal{M}$ consists of a set $D_{\mathcal{M}}$ called the domain of $\mathcal{M}$, a meaning $\mathcal{M}(c)$ for each constant c where $\mathcal{M}(c) \in D_{\mathcal{M}}$, a meaning $\mathcal{M}(f)$ for each function symbol f such that if f has arity n then $\mathcal{M}(f)$ is a function taking n arguments in $D_{\mathcal{M}}$ and returning an element of $D_{\mathcal{M}}$, and finally a meaning $\mathcal{M}(P)$ for each predicate symbol P such that if P has arity n then $\mathcal{M}(P)$ is a function taking n arguments in $D_{\mathcal{M}}$ and returning a truth value (either T or F). Meaning can be assigned to terms and formulas if, in addition to the meanings provided by the model, one is also given a meaning for each variable. A variable interpretation ρ relative to a structure $\mathcal{M}$ is a mapping from variables to elements of $D_{\mathcal{M}}$. For any structure $\mathcal{M}$ and variable interpretation ρ relative to $\mathcal{M}$, and any term t, we can define the value of the term t under $\mathcal{M}$ and ρ, which I will denote as $V(t,\ M,\ \rho)$, by the following equations.

$$\begin{array}{rcl} V(c,\ \mathcal{M},\ \rho) & = & \mathcal{M}(c) \\ V(x,\ \mathcal{M},\ \rho) & = & \rho(x) \\ V(f(t_1,\ \ldots,\ t_n),\ \mathcal{M},\ \rho) & = & \mathcal{M}(f)(V(t_1,\ \mathcal{M},\ \rho),\ \ldots,\ V(t_n,\ \mathcal{M},\ \rho)) \end{array}$$

As an example, let $\mathcal{N}$ be a structure whose domain $D_{\mathcal{N}}$ is the integers, that interprets the binary function symbols "+" and "$*$" as addition and multiplication respectively, and that interprets numerical constants such as 2 and 7 as the appropriate integer in $D_{\mathcal{N}}$. For any variable interpretation ρ and domain element d we let $\rho[x := d]$ be the variable interpretation identical to ρ except that it maps the variable x to the value d. As an example of the semantic evaluation of terms we have the following.

$$V(+(*(2,\ x),\ 3),\ \mathcal{N}, \rho[x := d]) = 2d + 3$$

While the value of a term is a domain element, the value of a formula is a truth value. For a given structure $\mathcal{M}$, and variable interpretation ρ relative to $\mathcal{M}$, the value of formulas is defined by the following equations.

$$\begin{array}{rcl} V(P(t_1,\ \ldots,\ t_n),\ \mathcal{M},\ \rho) = T & \text{iff} & \mathcal{M}(P)(V(t_1,\ \mathcal{M},\ \rho),\ \ldots,\ V(t_n,\ \mathcal{M},\ \rho)) \\ V(t_1 = t_2,\ \mathcal{M},\ \rho) = T & \text{iff} & V(t_1,\ \mathcal{M},\ \rho) = V(t_2,\ \mathcal{M},\ \rho) \\ V(\Phi_1 \vee \Phi_2,\ \mathcal{M},\ \rho) = T & \text{iff} & V(\Phi_1,\ \mathcal{M},\ \rho) = T \text{ or } V(\Phi_2,\ \mathcal{M},\ \rho) = T \\ V(\Phi_1 \wedge \Phi_2,\ \mathcal{M},\ \rho) = T & \text{iff} & V(\Phi_1,\ \mathcal{M},\ \rho) = T \text{ and } V(\Phi_2,\ \mathcal{M},\ \rho) = T \\ V(\Phi_1 \rightarrow \Phi_2,\ \mathcal{M},\ \rho) = T & \text{iff} & V(\Phi_1,\ \mathcal{M},\ \rho) = F \text{ or } V(\Phi_2,\ \mathcal{M},\ \rho) = T \\ V(\neg\Phi,\ \mathcal{M},\ \rho) = T & \text{iff} & V(\Phi,\ \mathcal{M},\ \rho) = F \\ V(\forall x \Phi,\ \mathcal{M},\ \rho) = T & \text{iff} & \text{for all } d \in D_{\mathcal{M}},\ \ V(\Phi,\ \mathcal{M},\ \rho[x := d]) = T \\ V(\exists x \Phi,\ \mathcal{M},\ \rho) = T & \text{iff} & \text{there exists } d \in D_{\mathcal{M}}\ \ s.t. V(\Phi,\ \mathcal{M},\ \rho[x := d]) = T \end{array}$$

Inference is a computational process for determining symbolic consequences of symbolic assertions. For example, consider the following formula.

$$\forall x, y, z\ \ (x \leq y \wedge y \leq z) \rightarrow x \leq z \tag{1}$$

Here $\forall x, y, z\ \Phi$ is being used as an abbreviation for nested universal quantification and $x \leq y$ is being used as a more legible notation for the formula

$\leq (x,\ y)$ where $\leq$ is a binary predicate symbol. Since $\leq$ is a binary predicate symbol, in any structure $\mathcal{M}$ we must have that $\mathcal{M}(\leq)$ is a binary relation on the set $D_{\mathcal{M}}$. Formula (1) is true in structure $\mathcal{M}$ provided that $\mathcal{M}(\leq)$ is transitive. Note that all variables in (1) are quantified — we say that all variable occurrences are *bound*. A variable occurrence not bound by a quantifier is called a *free occurrence*. A formula with no free occurrences of variables is called *closed*. The truth value of a closed formula (such as formula (1)) is determined by the structure and is independent of the variable interpretation.

Formula (1) is true in a structure whose domain is the natural numbers and which interprets the predicate symbol $\leq$ as the standard ordering predicate on natural numbers. However, formula (1) is false in other structures. We say that formula Φ *entails* formula Ψ, written $\Phi \models \Psi$, if, for any pair $\langle \mathcal{M},\ \rho \rangle$ such that $V(\Phi,\ \mathcal{M},\ \rho) = T$ we have $V(\Psi,\ \mathcal{M},\ \rho) = T$. If Σ is a set of formulas then we say that a pair $\langle \mathcal{M},\ \rho \rangle$ satisfies Σ if $V(\Phi,\ \mathcal{M},\ \rho) = T$ for every Φ in Σ. We say that Σ entails Ψ, written $\Sigma \models \Phi$, if for every pair $\langle \mathcal{M},\ \rho \rangle$ satisfying Σ we have $V(\Psi,\ \mathcal{M},\ \rho) = T$. As we shall see in later sections, for a given set Σ of formulas representing "knowledge" of some sort, and given question formula Ψ, symbolic processing (theorem proving) can, in many cases, determine whether $\Sigma \models \Phi$. Now we turn to the significance of this symbolic processing in early logicist models of AI.

4 The Situation Calculus

In logicist models of artificial intelligence, human world knowledge is modeled by a large set of first order formulas [7]. These formulas are taken to express "common sense knowledge". A fundamental concept in early AI models is the notion of a situation. Intuitively, a situation is certain place at a certain time. For example, the coffee room in the mathematics department at about 4pm on a certain day. In addition to having variables that range over situations, early AI models often used variables and terms intended to denote actions. An action is something that an agent can choose to do. The formula $\mathtt{CAN}(x, a, s)$ can be used to represent the statement that in situation s agent x can choose to take action a. The term $\mathtt{result}(x,\ a,\ s)$ can be used to represent the situation that results from agent x performing action

a in situation s. The formula $\texttt{ON}(x,\ y,\ s)$ can be used to represent the assertion that object x is on object y in situation s. An example of a formula expressing world knowledge might be the following.

$$\forall x, y, z, s\ \ \texttt{CAN}(x,\ \texttt{PUTON}(y,\ z),\ s) \rightarrow ON(y,\ z,\ \texttt{RESULT}(x,\ \texttt{PUTON}(y,\ z),\ s))$$

In the early logicist models symbolic inference from world knowledge is presumed to be the computational method by which people "understand" the world, i.e., the way they make predictions, answer questions, and plan actions.

5 Resolution Theorem Proving

While section 3 defines semantic entailment for first order formulas it does not define any computational process. Early AI research focused on the resolution principle as a means of mechanizing general purpose symbolic inference. Resolution is based on the notion of a clause. Formula (1), expressing the transitivity of the predicate $\leq$, is equivalent to the following.

$$\forall x, y, z\ \ \neg(x \leq y) \vee \neg(y \leq z) \vee x \leq z \tag{2}$$

To see the equivalence one should try to verify that in any model where (1) is true we have that (2) is true and vice-versa. Formula (2) is in a special form called a clause. More formally, an atom is defined to be a formula of the form $P(t_1,\ \ldots,\ t_n)$ and a literal is defined to be either an atom or the negation of an atom. The formula $x \leq z$ is an atom (which is a special case of a literal) and the formula $\neg(x \leq y)$ is a literal. A clause is defined to be a closed formula of the following form where each Φ_k is a literal.

$$\forall x_1, \ldots, x_n\ \ \Phi_1 \vee \Phi_2 \vee \ldots \vee \Phi_k \tag{3}$$

It turns out that, in a technical sense, clauses suffice for arbitrary first order knowledge representation and reasoning. To state the technical sense in which this is true I will introduce some notation. For closed formulas Φ and Ψ we have $\Phi \models \Psi$ if and only if there is no model $\mathcal{M}$ such that $V(\Phi,\ \mathcal{M}) = T$ but $V(\Psi,\ \mathcal{M}) = F$, or equivalently, the formula $\Phi \wedge \neg\Psi$ is unsatisfiable (is not true in any model). This means that any question of entailment can be reduced to a question of satisfiability. The following lemma implies that any question of entailment can be reduced to a question about clauses.

Lemma 1 *Any closed first order formula Φ can be converted, in time linear in the size of Φ, into a finite set Σ of clauses such that Φ is satisfiable if and only if Σ is satisfiable (there is a model satisfying all of the clauses in Σ).*

I will not prove this lemma here, but the basic idea is to introduce predicates representing subformulas of Φ. For example, if $\Psi[x,\ y]$ is a subformula of Φ involving the variables x and y then one introduces a predicate P_Ψ and adds clauses to Σ ensuring that in any model of the clauses we have that $P_\Psi(x,\ y)$ is true if and only if $\Psi[x,\ y]$ is true. The clauses defining the predicate for $\Psi \wedge \Psi_2$ can be stated in terms of the predicates for Ψ_1 and Ψ_2. A similar construction applies to other formula types. When a clause representation is desired, world knowledge is often written directly as clauses and no mechanical translation is used. The existence of a mechanical translation shows, however, that clauses suffice as a representation language.

Resolution is a mechanical inference procedure for deriving new clauses from a given set of clauses. It can be viewed as composed of two inference rules — factoring and binary resolution (these two rules can be combined into a single rule at some loss of clarity). In discussing these rules I will not bother to write the universal quantifier in front of clauses but it should be understood that the variables in a clause are universally quantified as in formula (3).

The factoring rule can be applied to clauses where two of the literals in the clause can be interpreted as the same literal. For example, consider the clause $P(x,\ y) \vee P(y,\ x)$. By considering the case where $x = y$ we can derive the clause $P(x,\ x)$. To formally state the general factoring rule we first need to define some terminology. A substitution is a mapping from variables to terms. For any substitution σ and literal Φ we define $\sigma[\Phi]$ to be the result of replacing each variable x in Φ by $\sigma(x)$. Two literals Φ_1 and Φ_2 are unifiable if they can be interpreted as the same literal, i.e., there exists a substitution σ such that $\sigma[\Phi_1]$ is the same literal as $\sigma[\Phi_2]$. In this case σ is called a *unifier* of Φ_1 and Φ_2. If Φ_1 and Φ_2 are unifiable, then a *most general unifier* (mgu) of Φ_1 and Φ_2 is a unifier γ such that any other unifier σ is a specialization of γ, i.e., there exists a substitution δ such that for any variable x we have that $\delta[\gamma(x)] = \sigma(x)$. The most general unifier of two literals is unique up to an arbitrary renaming of variables in the range of the substitution and can be computed in time linear in the size of the two given literals. The general form of the factoring rule states that, given a clause $\Phi_1 \vee \ldots \vee \Phi_n$ containing two unifiable literals Φ_i and Φ_j, one can derive the clause $\gamma[\Phi_1] \vee \ldots \vee \gamma[\Phi_n]$

where γ is the mgu of Φ_i and Φ_j. The resulting clause always has fewer literals.

The other fundamental inference rule of resolution theorem proving is binary resolution. Here I will assume that the literals in a clause can be reordered arbitrarily (a clause is viewed as a set of literals). I will also assume that the variables in a clause can be renamed arbitrarily at any time (they are universally quantified). The binary resolution rule states that given two clauses

$$\Phi_1 \vee \cdots \Phi_{n-1} \vee \Phi_n$$

$$\neg\Psi_1 \vee \Psi_2 \cdots \vee \Phi_k$$

where no variable occurs in both clauses (the second clause can be renamed to make this true) and where Φ_n and $\neg\Psi_1$ are unifiable, one can derive the following where γ is the mgu of Φ_n and $\neg\Psi_1$.

$$\gamma[\Phi_1] \vee \cdots \vee \gamma[\Phi_{n-1}] \vee \gamma[\Psi_2] \vee \cdots \vee \gamma[\Psi_k]$$

As an example consider the simple statement of transitivity in formula (1). Converting this formula to the clause (2) and applying binary resolution between this clause and a renamed version of itself results in the following.

$$\neg(x \leq y) \vee \neg(y \leq z) \vee \neg(z \leq w) \vee x \leq w$$

This clause is equivalent to the following implication.

$$x \leq y \wedge y \leq z \wedge z \leq w \rightarrow x \leq w$$

So the resolution rule can be used to compose the transitivity clause with itself to yield a three-step transitivity statement.

As another very simple example, suppose we want to show that $P \wedge (P \rightarrow Q) \models Q$. Here P and Q are predicate symbols of no arguments, i.e., they are formula constants. This can be formulated as the problem of showing that the set of formulas $\{P,\ P \rightarrow Q,\ \neg Q\}$ is unsatisfiable. The formula $P \rightarrow Q$ can be replaced by the clause $\neg P \vee Q$. The binary resolution rule can be applied to the clause P and the clause $\neg P \vee Q$ to yield the clause Q. The binary resolution rule can then be applied to the clause Q and the clause $\neg Q$ to yield the empty clause. A derivation of the empty clause shows the given clause set is unsatisfiable and hence the original entailment holds. The resolution rules are complete in the following sense.

Theorem 2 *If Σ is an unsatisfiable set of clauses then the rules of factoring and binary resolution can be used to derive the empty clause.*

The proof of this theorem is somewhat involved and will not be given here — a complete proof can be found in [6]. Lemma 1 and theorem 2 show that resolution theorem proving is, in a technical sense, sufficient for first order inference.

Early researchers soon discovered that mechanical resolution theorem proving was in many cases impractical — while the empty clause was derivable in principle, the system might take the age of the universe to derive it. Over the years there has been much research into methods for making more efficient variants of resolution theorem proving. Today, resolution theorem provers are very good at solving certain problems particularly amenable to first order reasoning [6]. However, first order logic itself seems to be of limited value in the applications of logic mentioned in the introduction. Other logics seem more relevant.

6 The Boyer-Moore Logic

As mentioned in section 2, different logics differ in the way they associate values with expressions. One way of assigning values to expressions is inspired by computer programming. Most programming languages allow expressions which include procedure calls. For example, if f is a defined procedure then the expression $x + f(y)$ computes to the sum of the value of x and the value returned by the procedure f when applied to the argument y. As in programming languages, the Boyer-Moore logic requires function and predicate symbols (procedure names) to be *defined* — the precise input-output relation of each function and predicate symbol must be completely specified. This is not true of first order logic — in first order logic formula (1) can be assumed and used without stating any definition of the meaning of the predicate $\leq$. In the Boyer-Moore logic the specification of the meaning of a function symbol is directly analogous to the definition of a procedure in a programming language. In fact, the Boyer-Moore logic is a kind of programming language plus inference rules for proving general assertions such as $\forall x f(f(x)) = f(x)$ where f is a defined procedure. Note that no amount of testing formally establishes a desired property on all of the infinitely many possible inputs to a procedure. Establishing a statement for all inputs requires logical inference.

The Boyer-Moore logic and its descendent ACL2 [5] are complex languages with many features that have been used to prove for a large number of mathematical statements and hardware and software verifications. Here I will try to explain the essential ideas of this logic by presenting a simpler but similar logic which I will call the simplified Boyer-Moore logic (SBM). The terms of SBM are defined by the following grammar where $\texttt{"}s\texttt{"}$ ranges over character string constants, $\texttt{if}$, $\texttt{pair}$, Π_1, Π_2, $\texttt{stringp}$, and $\texttt{equal}$ are special function symbols (primitives of the language) and f ranges over procedure names.

$$\begin{aligned} t \;\; ::= \;\; & \texttt{"}s\texttt{"} \mid x \mid \texttt{if}(t_1,\ t_2,\ t_3) \mid \texttt{pair}(t_1,\ t_1) \mid \Pi_1(t) \mid \Pi_2(t) \\ & \mid \texttt{stringp}(t) \mid \texttt{equal}(t_1,\ t_2) \mid f(t_1,\ \ldots,\ t_n) \end{aligned}$$

The method of assigning values to terms differs from that of first order logic in that here the evaluation process must be given a definition for each procedure name. A definition is an expression of the form $f(x_1,\ \ldots,\ x_n) \equiv t$ where t is a term not containing variables other than $x_1, \ldots, x_n$. A set D of definitions will be called self-contained if every procedure symbol appearing on the right hand side (the body) of a definition is itself defined in D. Note that a self-contained set of definitions allows recursive definitions. For example, under the standard way of representing sequences with pairs we can define a procedure $\texttt{append}$ for concatenating two sequences as follows.

$$\texttt{append}(x,\ y) \equiv \texttt{if}(\texttt{stringp}(x),\ y,\ \texttt{pair}(\Pi_1(x),\ \texttt{append}(\Pi_2(x),\ y)))$$

For the remainder of this section we assume a give self-contained set of definitions. A term all of whose procedure names have definitions (in the given definition set) will be called *fully-defined.* Values are themselves taken to be terms defined by the following grammar.

$$v ::= \texttt{"}s\texttt{"} \mid \texttt{pair}(v_1,\ v_2)$$

In other words, a value is either a string constant or a pair of values. If t is a fully-defined closed term (one not containing variables) we say that t has value v, written $t \Rightarrow v$, if a certain evaluation process generates value v for t. This can be made more precise by giving specific inference rules for deriving values. For any string constant $\texttt{"}s\texttt{"}$ we have $\texttt{"}s\texttt{"} \Rightarrow \texttt{"}s\texttt{"}$. If

$t_1 \Rightarrow v_1$ and $t_2 \Rightarrow v_2$ then $\texttt{pair}(t_1,\ t_2) \Rightarrow \texttt{pair}(v_1,\ v_2)$. Note that these two rules together imply $v \Rightarrow v$ for any value v. If $t \Rightarrow \texttt{pair}(v_1,\ v_2)$ then $\Pi_1(t) \Rightarrow v_1$ and $\Pi_2(t) \Rightarrow v_2$. If $t \Rightarrow$ "s" then $\texttt{stringp}(t) \Rightarrow$ "true" and if $t \Rightarrow \texttt{pair}(v_1,\ v_2)$ then $\texttt{stringp}(t) \Rightarrow$ "false". If $t_1 \Rightarrow$ "true", and $t_2 \Rightarrow v$ then $\texttt{if}(t_1,\ t_2,\ t_3) \Rightarrow v$. If $t_1 \Rightarrow$ "false" and $t_3 \Rightarrow v$ then $\texttt{if}(t_1,\ t_2,\ t_3) \Rightarrow v$. If $t_1 \Rightarrow v$ and $t_2 \Rightarrow v$ then $\texttt{equal}(t_1,\ t_2) \Rightarrow$ "true". If $t_1 \Rightarrow v_1$ and $t_2 \Rightarrow v_2$ with v_1 different from v_2 then $\texttt{equal}(t_1,\ t_2) \Rightarrow$ "false". Finally if $f(x_1,\ \ldots,\ x_n) \equiv s$ is one of the given definitions, and $v_1, \ldots, v_n$ are values, we write $s[v_1/x_1,\ \ldots,\ v_n/x_n]$ for the result of replacing each variable x_i with the value v_i in s. If $t_1 \Rightarrow v_1, \ldots, t_n \Rightarrow v_n$; the procedure f is defined by $f(x_1,\ \ldots,\ x_n) \equiv s$ and $s[v_1/x_1,\ \ldots,\ v_n/x_n] \Rightarrow w$ then $f(t_1,\ \ldots,\ t_n) \Rightarrow w$. These inference principles do not always allow a value to be derived. For example, if f is defined by $f(x) \equiv \texttt{pair}(x,\ f(x))$ then no value can be derived for $f(\text{"}a\text{"})$. However, if a value can be derived for a term then that value is unique.

The given system of definitions is called *terminating* provided that a value can be derived for every fully defined closed term. The Boyer-Moore logic enforces the constraint that the given system of definitions must be terminating — a definition is only accepted after termination has been proved. The methods for proving termination in the Boyer-Moore logic involves well founded orders on values — details can be found in [5].

The SBM logic can express mathematical facts as universally quantified terms. For example, given the above definition of append, the following universally quantified statement holds.

$$\forall x, y, z\ \ \texttt{equal}(\texttt{append}(x,\ \texttt{append}(y,\ z)), \quad \texttt{append}(\texttt{append}(x,\ y),\ z))$$

In general a formula of the form $\forall x_1, \ldots, x_n\ t$ is true if, for all values $v_1, \ldots, v_n$, we have $t[v_1/x_1, \ldots, v_n/x_n] \Rightarrow$ "true". It is easy to give SBM definitions of the Boolean operations and, or, not and implies. It is also easy to define natural arithmetic operations on natural numbers and to formulate Fermat's last theorem as follows where and5 is a five argument conjunction operator.

$$\begin{array}{ll}\forall\ \mathrm{n,x,y,z} & \texttt{implies}(\texttt{and5}(\texttt{nat}(\mathrm{n}), \texttt{nat}(\mathrm{x}), \texttt{nat}(\mathrm{y}), \texttt{nat}(\mathrm{z}), \texttt{greater}(\mathrm{n}, \texttt{two}())), \\ & \quad \texttt{not}(\texttt{equal}(\texttt{sum}(\texttt{exp}(\mathrm{x}, \mathrm{n}), \texttt{exp}(\mathrm{y}, \mathrm{n})), \texttt{exp}(\mathrm{z}, \mathrm{n}))))\end{array}$$

It turns out that a wide variety of statements can be formulated as universally quantified terms of SBM. The Boyer-Moore logic and its descendent ACL2 have been used to state and prove a wide variety of theorems of number theory and to formulate and verify specifications for microprocessors, machine language assemblers and compilers. All of these theorems and specifications can be easily formulated (though not easily verified) in the simplified logic SBM presented here.

7 Definitions, Completeness, and Gödel's Theorem

Gödel's incompleteness theorem states that for any sufficiently expressive logic there can not exist any sound and complete system of inference. Intuitively, sufficiently expressive means that the language is expressive enough to make simple statements about the integers, such as the statement that a certain polynomial does not equal zero for any integer values of its variables. The logic SBM is clearly expressive enough to make these kinds of statements. It follows that no sound and complete inference system exists for SBM. Here sound means that every provable statement is true and complete means that every true statement is provable. In practice one works with inference rules that are sound but incomplete. Although no precise characterization has been given of the statements provable using the inference rules provided by the Boyer-Moore system, it is clear that these inference rules are powerful enough to prove much of number theory and most specifications for hardware and software systems.

Resolution theorem proving provides a sound and complete system of inference for first order logic. It follows immediately from Gödel's incompleteness theorem that first order logic is not sufficiently expressive. In particular, first order logic is not expressive enough to define addition and multiplication on the natural numbers — the natural numbers themselves can not be defined in first order logic. More specifically, for any set of first order statements about the natural numbers there exist "nonstandard" models of those statements — models different from (not isomorphic to) the natural numbers in which those statements are also true.

It is possible to introduce a notion of "nonstandard model" for the Boyer-Moore logic and replace the notion of truth with the notion of being true in all possible nonstandard models. This reduces the expressive power of the logic — one can no longer define the set of natural numbers and the standard arithmetic operations on them. However, it allows a complete system of inference to be given — all statements that are true in all the nonstandard interpretations can be proved. On the other hand, the natural semantics seems easier to understand and allows, in principle, the construction of ever more powerful proof systems. In practice the construction of ever more powerful proof systems is limited by the fact that all human proofs can be expressed in ZFC set theory and hence no proof system more powerful than ZFC can be proved sound by human mathematicians.

8 Higher Order Logic

Second order logic allows quantification over predicates on individuals. Third order logic allows quantification over predicates that take predicates as arguments. Higher order logic allows nth order quantification for any n. It turns out that higher order logic is of central importance in the theory of computer programming languages and in the study of the syntax and semantics of human language. To formally define higher order logic one must first define types. The type of an expression determines the order of that expression. For example, a predicate P of one argument has type $D \rightarrow B$. This means that it takes an individual (a first order object) as its input and returns a truth value as its output. One writes $P : D \rightarrow B$ to indicate that P has type $D \rightarrow B$. One can also write $x : D$ to indicate that x is a first order value or write $f : (D \rightarrow B) \rightarrow B$ to indicate that f takes a predicate as an argument and returns a truth value. More generally one writes $f : \tau$ to indicate that f has type τ where τ is a type expression as defined by the following grammar.

$$\tau ::= D \mid B \mid \tau_1 \rightarrow \tau_2$$

Here D is the type of individuals (first order values) and B is the type of truth values and $\tau_1 \rightarrow \tau_2$ is the type of functions which take as input values of type τ_1 produce as output values of type τ_2. For each type expression we assume an infinite number of variables and constants of that type. I will write x^τ and c^τ to signify variables and constants of type τ. The terms of higher order logic are defined by the following grammar.

$$t ::= x^\tau \mid c^\tau \mid \texttt{apply}(t_1,\ t_2) \mid \lambda(x^\tau,\ t)$$

The term $\texttt{apply}(t_1,\ t_2)$ represents the application of the operator t_1 to the argument t_2. The term $\lambda(x^\tau,\ t)$ represents the function that takes an input of type τ and produces as output the value of t when x^τ is assigned the input value. We write $t : \tau$ to indicate that the term t is a well-typed term of type τ. This well-typedness relation is defined by certain simple rules. First, we have $x^\tau : \tau$ and $c^\tau : \tau$. Next, if $t_1 : \sigma \rightarrow \tau$ and $t_2 : \sigma$ then $\texttt{apply}(t_1,\ t_2) : \tau$. Finally, if $t : \tau$ then $\lambda(x^\sigma,\ t) : \sigma \rightarrow \tau$. A term t is called well-typed if there exists a type τ such that these rules derive $t : \tau$. Note that if t is well typed then there is only one type τ such that $t : \tau$.

Just as there are many logics, there are many different higher order type systems. The system presented above is called the simply typed lambda calculus with explicit types. In other systems the same variable might be assigned different types and the single term $\lambda(x,\ x)$ can be assigned any type of the form $\tau \rightarrow \tau$. Still other systems allow type schemes and quantified types but these are beyond the scope of this review.

A variety of abbreviations are standard in treatments of higher order logic. The term $\texttt{apply}(t_1,\ t_2)$ is usually abbreviated as $(t_1\ t_2)$. Types are usually left implicit on variables and constants so that x denotes a typed variable. The λ-term $\lambda(x^\tau,\ t)$ is often written as $\lambda x.\ t$. It turns out that functions of more than one argument can be represented by functions of one argument. The term $\lambda x, y.\ t$ is used as an abbreviation for $\lambda x. \lambda y. t$. The notation $\sigma_1 \times \sigma_2 \rightarrow \tau$ is taken to be an alternate notation for $\sigma_1 \rightarrow (\sigma_2 \rightarrow \tau)$ and the two argument application $(f\ x\ y)$ is taken to be an abbreviation for $((f\ x)\ y)$.

In higher order logic we assume the standard logical Boolean constants $\vee : B \times B \rightarrow B$, $\wedge : B \times B \rightarrow B$, $\neg : B \rightarrow B$, and $\rightarrow : B \times B \rightarrow B$ (we rely on context to distinguish Boolean implication from the type-theoretic function constructor). These constants are assigned there standard logical meaning. In higher order logic we also assume constants representing existential and universal quantification so that $\forall x^\tau \Phi$ (where $\Phi : B$) can be taken to be an abbreviation for $\forall_\tau(\lambda x^\tau \Phi)$ where $\forall_\tau$ is a constant with the appropriate standard meaning of type $(\tau \rightarrow B) \rightarrow B$ Similarly $\exists x^\tau \Phi$ is taken to be an abbreviation for $\exists_\tau(\lambda x^\tau \Phi)$ where $\exists_\tau$ is an appropriate logical constant.

As in first order logic, the value of a well-typed higher order term is defined relative to a model where the model specifies a domain — a set of first order values — and a meaning for each nonlogical constant. Given a domain of first order values, each type expression is associated with a well defined set — D denotes the set of first order values; B denotes the set $\{T,\ F\}$; and $\tau_1 \to \tau_2$ denotes the set of (all semantic) functions that take as input an element of the set denoted by τ_1 and produce as output an element of the set denoted by τ_2. The value for each nonlogical constant c^τ must be an element of the set denoted by τ. A variable environment ρ assigns a value to each variable such that the value assigned to x^τ is in the set denoted by τ. Given a model $\mathcal{M}$, a variable environment ρ, and a well-typed term t, the value $V(t,\ \mathcal{M},\ \rho)$ can be defined in a manner similar to that used for first order logic. If we have $t : \tau$ then $V(t,\ \mathcal{M},\ \rho)$ is an element of the set denoted by τ.

The formalism of higher order logic and its uses in the the study of the syntax and semantics of human language are described in more detail in [3].

9 Higher Order Logic and Gödel's Theorem

Although higher order logic does not explicitly support recursive definitions, higher order logic is sufficiently expressive for Gödel's theorem to apply. In higher order logic one can write the following induction principle for the natural numbers.

$$\forall P \qquad [P(0) \wedge (\forall x\ P(x) \to P(x+1))] \to \forall x\ P(x)$$

Combined a few more simple axioms, this formula yields a set of formulas (the second order Peano axioms) whose only model (up to isomorphism) is the natural numbers. So no sound system of inference for higher order logic can be complete. As with the Boyer-Moore logic, it is possible to give an "unnatural" semantics making sound and complete inference possible. However, the natural semantics seems easier to understand and in principle allows for the construction of ever more powerful inference systems.

It is possible to approximate higher order logic using first order schemas. For example, the induction principle for natural number can be expressed by the infinite set of all first order formulas of the following form where $\Phi[x]$ represents an arbitrary first order formula with a single free variable x.

$$[\Phi[0] \wedge (\forall x\ \Phi[x] \to \Phi[x+1])] \to \forall x\ \Phi[x]$$

This, however, is only an approximation to the second order statement given above — there are subsets of the integers that are nameable in richer (higher order) languages that are not nameable by a first order formula $\Phi[x]$. Hence there are uses of the true second order induction principle which are not covered by the first order schema. There are natural statements of number theory which are known to be true (they are provable in ZFC set theory) but are not provable from the first order version of Peano's axioms. The completeness theorem for first order logic implies that if Φ is not provable in first order Peano arithmetic then there must then be nonstandard models of these axioms where Φ is false, even if Φ is true of actual natural numbers. In the case of Peano arithmetic it seems reasonable to work with the higher order formulation even if this formulation is sufficiently expressive for Gödel's theorem to apply.

10 Summary

Logic is fundamental to our understanding of mathematics, computer programming, human language, and knowledge representation. Logic has long been proposed as a foundation for artificial intelligence. Although no intellectual framework has lead to the construction of machines matching human thought, it seems clear that logic remains relevant to the study of intelligence. It is not possible in this survey to due justice to the wide variety of logics and their applications. Perhaps most significantly, I have not addressed the relationship between logic and heuristic or statistical reasoning. The relationship between logic and statistics is currently an active area of research [4]. I can only hope that the next few decades will yield unambiguous advances in our understanding of intelligence.

References

[1] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1997.

[2] John Bell and Moshe Machover. *A Course in Mathematical Logic.* North-Holland, 1977.

[3] Bob Carpenter. *Type-Logical Semantics*. MIT Press, 1997.

[4] L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*. Springer-Verlag, 2001.

[5] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic, 2000.

[6] Alexander Leitsch. *The Resolution Calculus*. Springer-Verlag, 1997.

[7] Vladimir Lifschitz, editor. *Formalizing Common Sense: Papers by John McCarthy*. Intellect, 1998.