A VIEW OF AUTOMATED PROOF CHECKING AND PROVING

Christian LENGAUER

Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712-1188, U.S.A.

In memoriam, Dr. Martin Bartusch
** 24 July 1948*
† 19 July 1986

Abstract

Different techniques of automated formal reasoning are described and their performance and requirements on the human user are evaluated. The main trade-off is between autonomy and flexibility in conducting proofs. Examples of the use of techniques and existing systems are given, but not attempt of an exhaustive overview is made. The goal is to provide the reader with an idea of what to look for when selecting an approach for his/her application.

Keywords: Automated theorem proving, proof checking

1. Introduction

We use computers because we expect them to assist us in tackling problems. No matter what particular task we want the computer to assume, we expect it to increase the precision, or the speed, or the convenience with which we treat a problem. Frequently, we have several of these expectations at the same time.

Computers are electronic automata. As such, they are inherently formal. They are most reliably described and used in a formal manner. Humans are not automata or, if they are, the formal rules by which they operate are beyond our comprehension. Humans feel most comfortable with informal treatments.

How can we consolidate the different natures of the computer and the human?

(1) We can either cater to the human, and permit him/her to deal with the computer in an informal manner, even though it is an automaton. The consequence is a loss of precision, because the formal rules by which the computer operates are not understood. But we hope for a gain in convenience.

(2) Or we can cater to the computer and force the human to deal with it completely formally. The consequence is a loss of convenience. But we hope for a gain in precision.

Let us now constrain ourselves to the particular application of interest: automated reasoning. Our previous contemplations lead to two alternative routes of attack:

(1) *Informal reasoning.* This approach tries to assist the human in dealing with the computer in informal terms. Examples are techniques of machine learning, natural language understanding, and expert systems.

(2) *Formal reasoning.* This approach tries to assist the human in dealing with the computer in formal terms. It employs formal logics in which propositions can be expressed and reasoned about in a precise manner. These logics are implemented as model checking, proof checking, or theorem proving programs. Theorem proving programs can, again, be viewed as expert systems.

What do we want the computer to enhance–precision or convenience? Let us proceed on the assumption that precision is far more important to us than convenience, and that the problem at hand can be expressed in a formal logic. Then the approach we would choose is formal reasoning. We offer a brief survey of different techniques available today by which a computer can assist a human in justifying a formal claim, i.e., in proving a theorem.

2. Automated formal reasoning

Formal reasoning is the deduction of formally expressible propositions from other formally expressible propositions by way of formal inference rules. [1] In automated formal reasoning, these inference rules are applied by a computer program, a *formal reasoning system.* [2] We desire two properties of a formal reasoning system:

(1) *Consistency.* Only valid propositions can be proved.

(2) *Completeness.* All valid propositions can be proved.

A proposition is *valid* if it is a theorem in every model of the logic that the formal reasoning system implements. Consistency is absolutely essential. If the truth and untruth of one proposition cannot be distinguished, the truth and untruth of any proposition cannot be distinguished. Completeness is not as essential, and is often not attained by formal reasoning systems.

Model checking is a simple version of proving. *Model checking* means certifying the validity of a proposition in a proposed model. *Proving* means certifying the validity of a proposition independently of the model, i.e., in all possible models. For applications that are tied to specific models of interest, model

[1] This does not mean that propositions have to reflect facts that are known with certainty. Special logics that can express beliefs permit reasoning with uncertainty [19,24,33].

[2] We refer with the word "system" always to an implementation.

checking may be preferable to proving, because it can usually be made completely autonomous. For example, model checking has become popular for certifying claims in temporal or dynamic logics [7,14].

We shall here focus on proving and disregard model checking. Bundy [5] provides a good first introduction to automated theorem proving.

3. Proof procedures

At first sight, the best of all systems seems to be one that is consistent, complete, and entirely autonomous. We call such a system a *proof procedure*. Proof procedures are especially attractive for certain very limited application domains. For more general domains, they tend to be inefficient, because there is no freedom of choice in proof strategies–every proof is approached in the same way. To avoid such inefficiencies, one would need to take the particular nature of the proposition at hand into account. This is called *natural deduction* and is the topic of the next section.

We discuss, by example, two proof procedures: resolution, the (proof by) refutation procedure for first-order predicate logic, and Wu's Algorithm, a proof procedure for a part of elementary geometry.

3.1. RESOLUTION

Resolution is a consistent and complete proof procedure for first-order predicate logic. It is based on the principle of *refutation*: inferring a contradiction from the negation of the proposition to be proved.

3.1.1. Example [3]

Consider the following claim. Accepting the following hypotheses,
(1) whoever can read is literate,
(2) dolphins are not literate,
(3) some dolphins are intelligent,
we would like to conclude that
(4) some who are intelligent cannot read.
Our first step is to formalize the proposition in first-order predicate logic. We define four predicates:
$R(x)$ means x can read,
$L(x)$ means x is literate,
$D(x)$ means x is a dolphin,
$I(x)$ means x is intelligent.

[3] Borrowed from [20].

The logical formulas that correspond to our four propositions are:

(1) $(\forall x\colon R(x) \Rightarrow L(x))$,

(2) $(\forall x\colon D(x) \Rightarrow \neg L(x))$,

(3) $(\exists x\colon D(x) \wedge I(x))$,

(4) $(\exists x\colon I(x) \wedge \neg R(x))$.

The claim is $[(1) \wedge (2) \wedge (3)] \Rightarrow (4)$. Resolution works by refutation, that is, attempts to derive a contradiction from the negated claim $[(1) \wedge (2) \wedge (3) \wedge \neg(4)]$. Let us spell out the negation of (4):

(4′) $(\forall x\colon \neg I(x) \vee R(x))$.

Resolution can deal only with propositions that are in a syntactic normal form, the so-called *conjunctive normal form* or *clausal form*. A predicate in clausal form consists of groups of literals. A *literal* is a predicate symbol, or a negated predicate symbol. A group of literals is called a *clause*. Within each clause, literals are composed by disjunction ($\vee$). Clauses are composed by conjunction ($\wedge$). Any first order predicate can be put into clausal form. [4] Converting our propositions into clausal form yields:

(1) $\neg R(x) \vee L(x)$,

(2) $\neg D(x) \vee \neg L(x)$,

(3a) $D(a)$,

(3b) $I(a)$,

(4′) $\neg I(x) \vee R(x)$.

Imagine an unwritten $\wedge$ between any two neighboring clauses. Imagine also a $(\forall x\colon \ldots)$ around every clause that mentions x. a denotes an arbitrary constant in the range of variable x. A constant like a that is used to eliminate existential quantifiers is called a *skolem constant*.

To expose a contradiction in these five clauses, we employ the principle of resolution to derive new clauses. Take, for instance, clauses (1) and (2). They contradict each other in the validity of predicate $L(x)$. By the Law of the Excluded Middle, exactly one of $L(x)$ and $\neg L(x)$ must be valid. If $\neg L(x)$ is valid, then the validity of clause (1) requires the validity of $\neg R(x)$. If $L(x)$ is valid, then the validity of clause (2) requires the validity of $\neg D(x)$. Hence, clauses (1) and (2) let us infer the new clause:

(5) $\neg R(x) \vee \neg D(x)$.

The continuation of this resolution process is depicted as a layered graph whose nodes are clauses (fig. 1). The top layer of the graph contains the input clauses. Input clauses have no predecessors (nodes in higher layers). Each subsequent

[4] An algorithm to that effect is presented in [20].

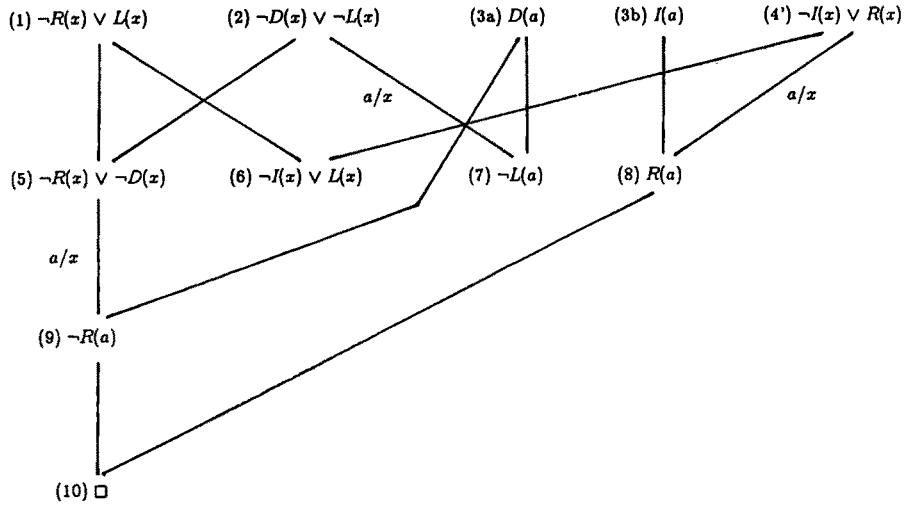

Fig. 1. A (partial) resolution graph.

layer contains the clauses obtained by resolving clauses of higher layers with each other. Each derived clause has two predecessors: the clauses it is derived from.

Clause (1) cannot be resolved with clauses (3a) or (3b) because it does not share literals with them, but clauses (1) and (4′) lead to the new clause (6). Clause (2) resolves with clause (3a) to clause (7). To derive clause (7), we must restrict the range of x to a. This process is called *unification*. Clause (2) does not resolve with clauses (3b) or (4′). Clause (3a) does not resolve with clauses (3b) or (4′). Clause (3b) resolves with clause (4′) to clause (8). At this point, we have resolved all input clauses against each other, i.e., we have completed the layer below the input clauses. We now proceed to resolve input clauses against derived clauses. Clauses (1) and (2) do not resolve with clause (5), but clause (3a) resolves with clause (5) to clause (9). At this point, we recognize a contradiction between the two most recent clauses and, resolving them, derive the empty clause (10). This terminates the proof.

Note that we proceeded top-down or *breadth-first* in the graph, resolving clauses with low ordinals before clauses with high ordinals. [5] Contradictions are derived much faster *depth-first*, resolving in every step the most recently derived clause with some other clause. The depth-first resolution proof of the previous claim requires only four steps. Unfortunately, depth-first resolution is not necessarily complete: we might get caught in a loop and bypass the contradiction by an infinite derivation of clauses.

[5] We made an exception in the last step to prevent the proof exposition from dwelling on.

3.1.2. Assessment

Resolution was the first popular method of automated theorem proving and a very promising start. It was invented by Robinson [25]. Unfortunately, resolution suffers from some major drawbacks–apart from the fact that the principle of proof by refutation is still debated in the mathematics community.

A resolution proof does not apply domain-dependent knowledge or exploit the structure of the proposition to be proved. In fact, its first move is to get rid of the structure by converting the proposition into clausal form. For instance, if a contradiction is found, resolution cannot reveal whether the contradiction arose between the hypotheses and the conclusion or already in the hypotheses themselves. [6] This obliteration of structure provides for little documentary value should the proof ever fail.

A proof by resolution does not proceed in a goal-directed manner. It simply amasses clauses that can be inferred from previous clauses. It is easy to generate a lot of clauses that will not contribute to the detection of a contradiction. To avoid that, a great number of control strategies and constrained resolution principles have been proposed [30]. Breadth-first and depth-first are examples of control strategies.

The resolution prover AURA (AUtomated Reasoning Assistant) of Wos and his colleagues incorporates many of these principles and strategies and a large number of switches with which the program can be "retuned" between proofs. AURA has helped in the solution of open problems in various fields of mathematics, logical circuit design, and chemical synthesis [28,29,31]. AURA's way to attack the efficiency problems of proof procedures is, essentially, to give the user a choice among several proof procedures. In the next subsection, we surmount the same problems by restricting the application domain.

3.2. WU'S METHOD

Hilbert [15] based elementary geometry on five groups of axioms:
(1) existence and incidence,
(2) order,
(3) congruence (or motion),
(4) parallelism,
(5) continuity.

Tarski [26] provided a proof procedure for propositions in elementary geometry, but it is hopelessly inefficient. The Chinese mathematician Wu [32] was able to devise a practical proof procedure by eliminating axioms (2) and (5). In his restricted version of elementary geometry, one can talk about circular or straight

[6] In other words, resolution is *refutation-complete*, i.e., it will expose a contradiction if there is one, but it is not *deduction-complete*, i.e., it cannot determine whether one proposition is the logical conclusion of a set of other propositions [30].

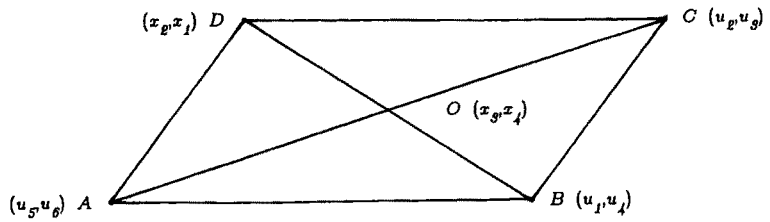

Fig. 2. A parallelogram.

lines and cross or end points, but one cannot talk about positions on lines *between* cross or end points. Chou [6] has implemented Wu's algorithm and obtained a system that can efficiently and independently prove non-trivial theorems in elementary geometry.

3.2.1. Example [7]

Let us convey the geometrical situation that we consider by a picture (fig. 2). Our claim is about the points of a parallelogram. We claim that the length of the line from point A to point O equals the length of the line from point C to point O, i.e., $|AO| = |CO|$. The parallelogram is positioned arbitrarily on the plane. Its position is determined by the coordinates of any three points that are not on the same line, here, A, B, and C.

Wu's idea was to transform geometrical situations into polynomial equations. [8] For example, our geometrical situation is expressed by the following equations:

$f1$: $(x_1 - u_3)(u_5 - u_1) - (x_2 - u_2)(u_6 - u_4) = 0$
$A = B$ or $C = D$ or $AB \parallel CD$

$f2$: $(x_1 - u_6)(u_2 - u_1) - (x_2 - u_5)(u_3 - u_4) = 0$
$A = D$ or $B = C$ or $AD \parallel BC$

$f3$: $(x_1 - u_4)(x_3 - u_1) - (x_2 - u_1)(x_4 - u_4) = 0$
O is on BD

$f4$: $(u_3 - u_6)(x_3 - u_5) - (u_2 - u_5)(x_4 - u_6) = 0$
O is on AC.

The claim is represented by the quadratic polynomial equation:

g: $(x_3 - u_5)^2 + (x_4 - u_6)^2 - (u_2 - x_3)^2 - (u_3 - x_4)^2 = 0 \quad |AO| = |CO|$.

The proof of the claim is established by solving $f1$, $f2$, $f3$, $f4$, and g as polynomials in the variable coordinates of D and O. More precisely, $f4$, $f3$, $f2$, and $f1$ are, in this order, divided into g. We can choose the coordinates of A, B,

[7] Borrowed from [6].

[8] This forced him to eliminate axiom groups (2) and (5). They would have introduced inequalities in the polynomial representation.

and C to simplify the solution. For example, [6] chooses the origin for A, i.e., $u_5 = 0$ and $u_6 = 0$, and the horizontal axis of the coordinate system for the line AB, i.e., $u_4 = 0$. The division process imposes restrictions on the remaining coefficients. The details are too involved to be reproduced here; see [6]. We arrive at the following restrictions:

(1)	$u_1 \neq 0$	AD and BC are not the same line,
(2)	$u_3 \neq 0$	AB and CD are not the same line,
(3)	$u_1 \neq 0$ and $u_3 \neq 0$	AC and BD are not parallel,
(4)	$u_2 \neq 0$	C is not on the vertical axis.

3.2.2. Assessment

Wu's method is an especially successful proof procedure. Complicated theorems are proved independently in seconds. Chou's system might be viewed as the start of a MACSYMA [18] for elementary geometry.

The similarity to resolution is striking. In Wu's method, polynomials assume the role of clauses in resolution: the polynomials representing the hypotheses are "resolved" with the polynomial representing the conclusion. Coefficient restriction assumes the role of unification. Again, the structure of the proposition is eliminated. Independence, together with the ability to provide a qualified answer to a claim, make implementations of resolution and Wu's method useful tools for answering questions. [9] Used in this mode, their proponents claim, these systems can effectively assist the human in identifying and solving problems.

4. Natural deduction

A natural deduction system is composed of two components:

(1) The *deductive component* comprises the set of consistent inference rules by which new propositions are derived from previous propositions.
(2) The *cognitive component* comprises decision strategies that determine which inference rule is applied at some proof step.

The deductive component is essential and can be found in each natural deduction system. It is largely responsible for the consistency of the system. The cognitive component is not essential. It may be very primitive, or it may be very sophisticated. The cognitive component approximates completeness. Its purpose is to minimize the need for human intervention in the mechanical proof. Typically, applications that require little human intervention are called *theorem*

[9] A programming language for question answering, based on the principle of resolution, is PROLOG [8].

proving, and applications that require a lot of human intervention are called *proof checking*. Where the borderline between the two lies depends on the good will of the user. It must be noted, however, that the degree of autonomy of the natural deduction system and, therefore, its classification as a proof checker or prover may depend on the problem at hand. Various natural deduction systems can perform, without aid, "small" proofs or proofs that rest on a theory in which they are an expert. But they need help with "bigger" proofs or proofs that rest on theories in which they are not an expert. A system with a strong cognitive component will be more autonomous but also less flexible, because proofs are forced to comply with the strategies of the cognitive component. Proof checking and proving are relatively general-purpose approaches. For restricted problem domains, the autonomy of proof checking and proving can be enhanced by an automatic generation of most or all of the input to the proof checker or prover. This approach, *proof script generation*, is also discussed in this section.

4.1. PROOF CHECKING

We give an example of a natural deduction system with a very primitive cognitive component. The system is called PRL (Proof Refinement Logic, read "pearl") and has been developed by Constable and his group [1]. PRL is a descendant of Edinburgh LCF (Logic for Computable Functions) [13]. Recently, the PRL system has been expanded and given a new name: Nuprl (read "new pearl") [10].

4.1.1. Example [10]

The claim to be proved is:

$$(\forall x,\ y \in \mathrm{Int}\colon (\exists z \in \mathrm{Int}\colon\ z = \max(x,\ y))).$$

A transformation of the expression $z = \max(x,\ y)$ into logic yields:

$$(\forall x,\ y \in \mathrm{Int}\colon (\exists z \in \mathrm{Int}\colon (x < y \Rightarrow z = y) \wedge (x \geqslant y \Rightarrow z = x))).$$

Let us sketch the proof of the proposition with a proof tree (fig. 3). The nodes of the tree are subgoals to be proved. The arcs are labelled with the PRL deduction rules that are applied to derive the subgoals. The human inputs the claim and the rules to be applied. The system automatically selects the target operator of each rule (in the figure, stated in parentheses beside the rule) and prompts the next subgoals. Let us explain informally the rules used in this example, level by level:

(1) The first rule, "consequence" (seq), applies the principle of divide-and-conquer. The human proposes the consideration of two cases, $x < y$ and $x \geqslant y$.
(2) The validity of the introduced premiss must be substantiated; this is done by a proof procedure for equality, lists, and integer arithmetic (arith). Also, the

[10] Adapted from [23].

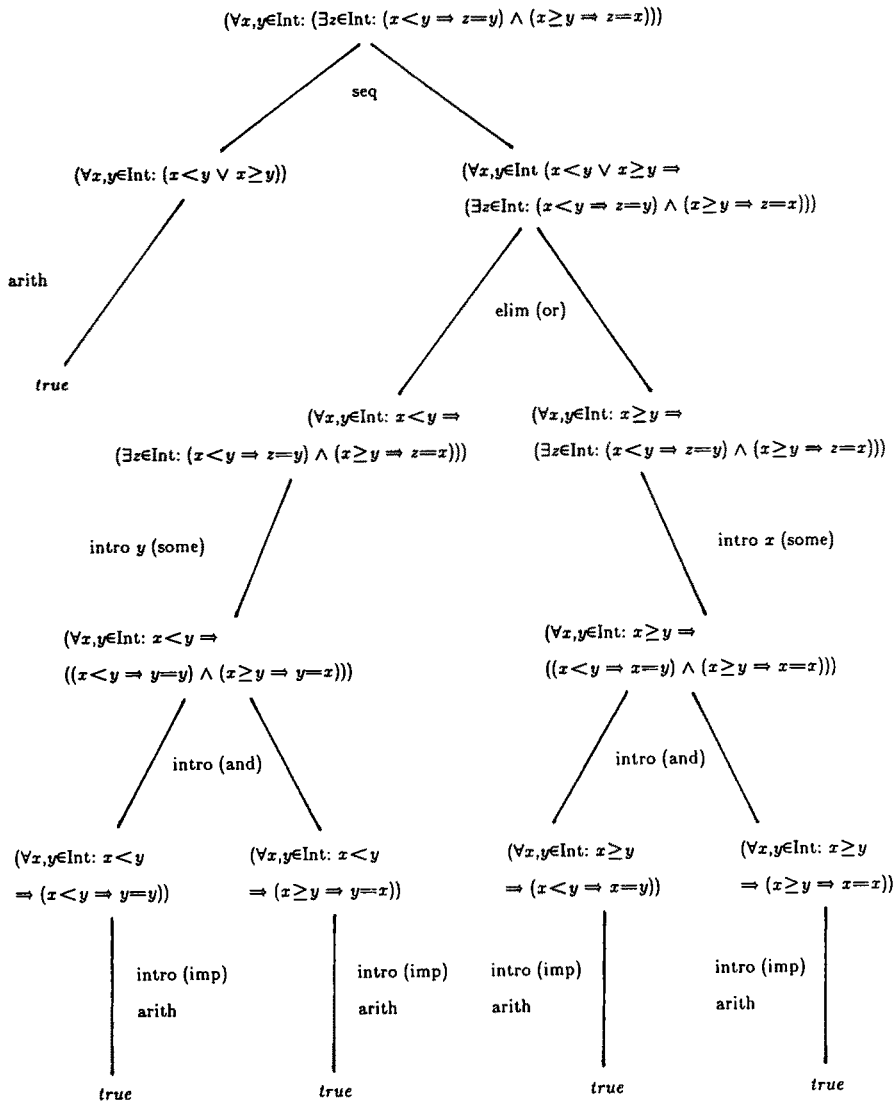

Fig. 3. A PRL proof tree.

two cases are split. The according rule is "elimination" (elim); the system autonomously identifies the ∨ in the premiss as the operator to be eliminated.

(3) The existence claims in the two subgoals are satisfied by proposing specific values for z via the "introduction" rule (intro).

(4) Introduction is applied, again, to split the conjuncts in the conclusion, generating four new subgoals.

(5) Introduction is applied, once more, to simplify the implications, and, finally, the proof procedure for arithmetic establishes the claimed equalities.

4.1.2. Assessment

The human largely has control over the proof (though, in the case of PRL, not completely, since the system selects the target operators of rules). The top-down proof log, which is produced with minimal input from the human, is of great documentary value. In the example, only system-defined rules were applied, but PRL also gives the human the choice of defining new rules in terms of previously defined rules. Such user-defined proof strategies are called *tactics*. Tactics are guaranteed safe because they are coded in a language, ML (the MetaLanguage), that does not introduce inconsistencies when combining consistent rules. Of course, tactics are not guaranteed to be useful. A good description of the use of tactics in PRL can be found in [9].

The main difference between PRL and its predecessor in spirit, LCF, is that the PRL system can automatically extract programs from proofs of propositions of the form:

(∀ inputs: (∃ outputs: predicate(inputs, outputs))).

The program extracted from the proof of our proposition is:

if $x < y$ **then** $z := y$ **else** $z := x$.

Large proofs that have been certified in this style deal with denotational semantics, functional programs, and digital circuits [21]. LCF and PRL can be viewed as kits for building programs that mechanically trace proofs: primitive deduction rules, like the ones applied in our example, are the basic building blocks and ML, the programming language for tactics, is the tool for building new blocks. Whereas proof checkers, like LCF and PRL, are tool kits for individual assembly, theorem provers, to be discussed in the next section, are already assembled tools.

4.2. THEOREM PROVING

We give an example of a natural deduction system with a significant cognitive component that is especially effective at performing induction proofs. It is called THM, or the Boyer-Moore prover, named after its two creators. The Boyer-Moore prover [2] employs several powerful heuristics: simplification by substituting equals for equals (using previously proved lemmas as rewrite rules), formula normalization (replacing destructors by constructors), "cross-fertilization" (eliminating equalities from the hypothesis after using them in the conclusion), induction (derived from an inspection of the recursive terms in the proposition), and–only after the first induction has been performed–a rather adventurous heuristic: generalization of the claim to be proved (replacing a term by a variable). The human interacts with the prover by

(1) submitting functions that enter in the formulation of a proposition,
(2) submitting axioms, i.e., propositions for whose validity he/she takes the responsibility, or
(3) submitting lemmas, i.e., propositions for proof by the system.

If a function is defined recursively, it will only be accepted if the prover can establish that the recursion is admissible, i.e., cannot lead to an infinite regress. The prover reports the progress of the proof through a natural language interface. When the proof is going badly, the human must abort it and try to steer the prover into another direction either by asking it to prove auxiliary propositions or by supplying specific proof hints like variable substitutions, specific lemmata already proved, induction schemes, etc.

4.2.1. Example

The Boyer-Moore mechanical logic is a functional logic. Predicates are functions with a boolean range. The most basic data structure is the list. Therefore, let us describe the proof of a simple property of lists. An *s-expression* is recursively defined as follows:

(1) An *atom* is an indestructible *s*-expression. It is denoted by some identifier.
(2) An *ordered pair* is a combination of two *s*-expressions x and y. It is denoted by $(x.y)$. A constructor function, *cons*, combines two *s*-expressions to an ordered pair; two extractor functions, *car* and *cdr*, extract the left and right element of an ordered pair, respectively.

A list is represented as an ordered pair whose left element is the first (left-most) element of the list, and whose right element is the list without its first element. For example, the list $(a\ b\ c)$ is represented is $(a.(b.(c.())))$. Every proper list must be a nested *s*-expression whose inner-most right element is $()$. $(a.(b.(c.d)))$ is not a proper list.

We can define two different functions that append *s*-expressions. The first function only appends proper lists properly: [11]

$$\begin{aligned} append(x, y) = \textbf{if } &\neg listp(x) \\ &\textbf{then } y \\ &\textbf{else } cons(car(x), append(cdr(x), y)) \end{aligned}$$

The functional predicate *listp* returns *true* if its argument is an ordered pair, and *false* otherwise. If the first argument is not a proper list, its inner-most right atom is dropped in the result, e.g., $append((a.b), (c.d)) = (a.(c.d))$. The following function works for all *s*-expressions:

$$\begin{aligned} append2(x, y) = \textbf{if } &\neg listp(x) \\ &\textbf{then if } x = () \\ &\qquad \textbf{then } y \\ &\qquad \textbf{else } cons(x, y) \\ &\textbf{else } cons(car(x), append(cdr(x), y)) \end{aligned}$$

[11] This is the traditional LISP append function [27].

For example, $append2((a.b), (c.d)) = (a.(b.(c.d)))$. A recognizer for proper lists is:

$$plistp(x) = \textbf{if } \neg listp(x) \textbf{ then } x = () \textbf{ else } plistp(cdr(x))$$

Plistp returns *true* if its argument is a proper list, and *false* otherwise. Finally, let us define a function, *all-atoms,* that eliminates the nesting structure of an *s*-expression. *All-atoms* returns a list that enumerates all atoms of its argument left to right:

$$all\text{-}atoms(x) = \textbf{if } \neg listp(x) \textbf{ then if } x = () \textbf{ then } () \textbf{ else } list(x) \textbf{ else } append(all\text{-}atoms(car(x)), all\text{-}atoms(cdr(x)))$$

For example, the results of $all\text{-}atoms(a)$, $all\text{-}atoms((a))$, and $all\text{-}atoms(((a)))$ are all (a). When the Boyer-Moore prover admits a function definition, it already remembers certain facts that it has discovered in the process of admission. For instance, when admitting function *all-atoms*, the prover stores for future reference that the result is either a literal atom (not a number) or an ordered pair. However, it is not clever enough to realize at this point that the ordered pair always represents a proper list.

Our claim is that *all-atoms* distributes over *append2*:

$$all\text{-}atoms(append2(x, y)) = append2(all\text{-}atoms(x), all\text{-}atoms(y))$$

The claim is in terms of the more general append function, *append2*. However, since *all-atoms* constructs proper lists only, the simpler *append* suffices in its definition. We will have to exploit the similarity of *append* and *append2*. We make the Boyer-Moore prover aware of this fact by asking it to prove two auxiliary propositions.

The first property is the identity of *append* and *append2* when their first argument is a proper list:

$$plistp(x) \Rightarrow (append2(x, y) = append(x, y))$$

The proof succeeds immediately by an induction (on the structure of *append2*).

The second property, that *all-atoms* returns proper lists only, enables the exploitation of the first property:

$$plistp(all\text{-}atoms(x)).$$

The proof proceeds, again, by induction (on the structure of *all-atoms*). But, in the course of the induction step, the prover discovers (by generalization) another

required property: that *append* of two proper lists returns a proper list. It proves this property autonomously with a second induction (on the structure of *plistp*).

With help of these two properties, the prover reduces our original claim to:

$$all\text{-}atoms(append2(x, y)) = append(all\text{-}atoms(x), all\text{-}atoms(y))$$

and then proceeds to prove this proposition independently. This proof is even more elaborate. Via two inductions (and generalizations), the prover works its way to the central property: the associativity of *append*, which it then proves with a third induction. Again, no intervention of the human is required.

4.2.2. Assessment

This is the Boyer-Moore prover in all its glory. The prover is not always as lucky at identifying appropriate inductions–and hardly ever at identifying appropriate generalizations. Even in this example, the two auxiliary lemmas are crucial. There are two ways of identifying needed intermediate lemmas: by working the proof on paper or by watching the prover go and stray. Typically one employs a mixture of both.

More involved claims quickly make the Boyer-Moore prover a lot more dependent on the human. Proofs often run into tens of auxiliary functions and hundreds of auxiliary lemmas. The prover has certified substantial claims in a variety of areas from elementary mathematics and metamathematics (including Goedel's incompleteness theorem), over real-time control, to the verification of software and hardware [4]. In all these proofs, the Boyer-Moore prover plays the role of a persistent and incorruptible colleague. Despite its sophistication, it requires a lot more precision and constructive input than a fellow human would expect.

4.3. PROOF SCRIPT GENERATION

Natural deduction proving requires a substantial amount of interaction with the human. This flexibility is much appreciated by people that conduct automated proofs. But sometimes, when we conduct many different proofs within the same confined theory, we would like our natural deduction system to be more independent. We might then elect to strengthen the heuristics of the system by adding a *proof script generator*: a preprocessor that converts our propositions into a sequence of commands for the proof system. Hopefully, this proof script will be complete enough to make any intervention during the mechanical certification unnecessary.

The most wide-spread application of this mechanical proof technique is *verification condition generation*. It is employed in the mechanical verification of programs. In this application, a proposition is of the form "program satisfies specification", where the program is expressed in some conventional programming language, and the specification consists of an annotation of the

program. The annotation consists of logical assertions about the program's variables. Assertion A placed before (after) statement S expresses that the values of the program's variables before (after) execution of statement S must satisfy condition A. Typically, a verification condition generator requires

(1) one assertion, the *precondition*, before the program statement that is to be executed first,
(2) one assertion, the *postcondition*, after the program statement that is to be executed last, and
(3) one assertion, the *loop invariant*, with every loop.

4.3.1. Example

We formulate the proposition that an iterative program returns the factorial of its input, n, as output, z. We denote the precondition by keyword **pre**, the postcondition by **post**, and the loop invariant by **inv**:

```
pre n ⩾ 0
z := 1;
if n ≠ 0
  then i := 1;
       inv z = i! ∧ i > 0
       while i ≠ n
             do
                i := i + 1;
                z := z * i
             od
fi
post z = n!
```

The verification conditions are generated by symbolic execution of the program code. In this example, four conditions are generated. The precondition and the negated condition of the if-statement must, when the value of z is 1 as the assignment preceding the if-statement specifies, imply the postcondition:

(1) $(n \geqslant 0 \wedge n = 0 \wedge z = 1) \Rightarrow (z = n!)$

The precondition and the condition of the if-statement must, when the values of z and i are 1, imply the loop invariant:

(2) $(n \geqslant 0 \wedge n \neq 0 \wedge z = 1 \wedge i = 1) \Rightarrow (z = i! \wedge i > 0)$

The loop invariant and the condition of the while-loop must, after execution of the loop body, again establish the loop invariant:

(3) $(z = i! \wedge i > 0 \wedge i \neq n) \Rightarrow (z(i+1) = (i+1)! \wedge i + 1 > 0)$

And the loop invariant together with the negated condition of the loop must establish the postcondition:

(4) $(z = i! \wedge i > 0 \wedge i = n) \Rightarrow (z = n!)$

These four propositions are submitted to the natural deduction system.

4.3.2. Assessment

We have come full circle: after first considering totally autonomous and inflexible proof techniques, we turned to elaborate dialogue and high flexibility, and then traced our steps back again to more and more autonomy and less flexibility. We have now gained back all the disadvantages with which we started out: proofs with no documentary value and no choice in proof strategy. We had better insist on high autonomy if we want to make the price worth-while! The success of proof script generation depends intimately on the type of propositions and the natural deduction system for which proof scripts are generated. Propositions must be easily decomposable. The behavior of the natural deduction system, at least in the domain for which proof script generation is contemplated, must be predictable enough to generate all required input.

While other attempts of proof script generation are rare, [12] verification condition generation has led to significant successes. Polak [22] specified formally the compilation problem for a Pascal-like language and employed verification condition generation to prove that his implementation observes the specification. Boyer and Moore have developed a generator of verification conditions in their logic for (a respectable subset of) FORTRAN programs [3]. Gypsy [12], a language for distributed programs, comes with a verification condition generator that can be backed up by a variety of theorem provers. The Gypsy environment has been used to verify several reasonably large programs, among them the most complex distributed program that has been verified to date: a special interface for the Arpanet [11]. This program is actually in use. Proofs with verification condition generation are typically based on a large set of axioms that describe basic properties of the programming language. This axiom set could be reduced by using an automated theorem prover to deduce some of the axioms from others, but builders and users of verification condition generators shy away from this time-consuming and tedious task. The verification condition approach really pays off, when the logical foundations on which it rests can be kept small.

5. Conclusions

Research efforts in formal automated reasoning focus primarily on two aspects: the efficiency and autonomy of the certification process, and the effectiveness and convenience of the dialogue between the human and the system. Roughly, formal automated reasoning is where electronic computing used to be 35 years ago: a

[12] For example, [17] exposes the so-called *Knuth-Bendix Problem* [16] as a potential hazard to proof script generation. A fact whose logical representation involves several functions can be expressed in many different ways depending on which function calls are expanded. Some of those representations will match hypotheses of rules or theorems known to the prover, others will not. The absence of human control in proof script generation makes it especially difficult to walk the tightrope of the prover's knowledge.

number of specialists can make very effective use of it, but it is not yet accessible to the general public. However, recent developments indicate a promising future. The technology of formal automated reasoning has already matured to the point of practical relevance for present-day applications. For example, the Department of Defense of the United States of America requires contract programming of a certain criticality to be certified by any one of a list of formal reasoning systems. The Gypsy verification environment is on that list, and the Boyer-Moore prover is part of another integrated system on that list.

It is not surprising that, so far, much of automated formal reasoning has been concerned with properties of computer software and hardware. Mechanical logics are nothing else but very restrictive and mathematically managable programming languages. For example, (depth-first search) resolution is, essentially, applicative PROLOG, and the Boyer-Moore logic is, essentially, applicative LISP. However, there is strong evidence, that formal automated reasoning can be applied to other formalizable domains as well. We have attempted to provide, in the previous sections, pointers to the relevant literature.

Often, the question is posed: "How can we trust the programs that check our proofs?" To date, no verified implementation of a formal reasoning system exists, but efforts are made to build these systems in layers, as is now common practice in operating system design. Each layer would be proved correct, assuming the layers below are correct. This way, only the most primitive layer must be trusted. Still, one should not expect infallibility from a formal reasoning system, but subject everyone of its claims to critical scrutiny. Here, natural deduction systems fare better than proof procedures, because of their more comprehensible proof documentation. Even if the implementation of a natural deduction system should commit an error–the decomposition of proofs into precise and, usually, small steps makes it likely that the human will catch it.

In the past, automated formal reasoning systems have been used mostly by the groups that built them or by their friends and associates. The question which approach to take did not arise. One chose the one that one had available. As relationships and trade-offs between different approaches emerge and systems become better documented and available on a larger variety of computers, one might attempt a more independent selection:

(1) Properly identify the problem domain. Take care to exclude irrelevant aspects. They could complicate reasoning unnecessarily.
(2) Try to develop a proof procedure that is efficient enough to be practical.
(3) If that does not succeed, try to find a formal reasoning system whose cognitive component is tailored towards your application. For example, if your proofs rely a lot on mathematical induction, you might select the Boyer-Moore prover. Try proof script generation.
(4) If you find that heuristics get in your way, turn to a formal reasoning system without cognitive component, like PRL. Tailor it for your specific purposes by incorporating your own tactics. Again, try proof script generation.

Acknowledgements

Thanks to Chua-Huang Huang and Bill Young for a careful reading. Presentations of Ray Bareiss on PRL and Bill Pierce on Gypsy in my *Formal Semantics and Verification* class helped shape this view. Financial support was provided by a grant from the Lockheed Missiles & Space Corporation.

References

[1] J.L. Bates and R.L. Constable, Proofs as programs, ACM TOPLAS 7, 1 (1985) 113–136.
[2] R.S. Boyer and JS. Moore, *A Computational Logic*, ACM Monograph Series (Academic Press, 1979).
[3] R.S. Boyer and JS. Moore, A verification condition generator for FORTRAN, in: *The Correctness Problem in Computer Science*, eds. R.S. Boyer and JS. Moore, International Lecture Series in Computer Science (Academic Press, 1981) pp. 9–102.
[4] R.S. Boyer and JS. Moore, Overview of a theorem prover for a computational logic, in: *8th Conf. on Automated Deduction*, ed. J.H. Siekmann, Lecture Notes in Computer Science 230 (Springer Verlag, 1986) pp. 675–678.
[5] A. Bundy, *The Computer Modelling of Mathematical Reasoning* (Academic Press, 1983).
[6] S.-C. Chou, Proving elementary geometry theorems using Wu's algorithm, in: *Automated Theorem Proving: After 25 Years*, eds. W.W. Bledsoe and D.W. Loveland, Contemporary Mathematics, Vol. 29 (American Mathematical Society, 1984) pp. 243–286.
[7] E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications: a practical approach, *Proc. 10th Ann. Symp. on Principles of Programming Languages*, 1983, pp. 117–126.
[8] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, 2nd ed. (Springer-Verlag, 1984).
[9] R.L. Constable, T.B. Knoblock and J.L. Bates, Writing programs that construct proofs, Journal of Automated Reasoning 1, 3 (1985) 285–326.
[10] R.L. Constable et al., *Implementing Mathematics with the Nuprl Proof Development System* (Prentice-Hall, 1986).
[11] D.I. Good, The proof of a distributed system in Gypsy, Tech. Rep. #30, Institute for Computing Science, The University of Texas at Austin, Sept. 1982.
[12] D.I. Good, Mechanical proofs about computer programs, in: *Mathematical Logic and Programming Languages*, eds. C.A.R. Hoare and J.S. Shepherdson, Series in Computer Science (Prentice-Hall Int., 1985) pp. 55–75.
[13] M.J.C. Gordon, A.J. Milner and C.P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science 78 (Springer Verlag, 1979).
[14] D. Harel, *First-order Dynamic Logic*, Lecture Notes in Computer Science 68 (Springer-Verlag, 1979).
[15] D. Hilbert, *Foundations of Geometry* (Open Court Publ. Co., 1971); Revised by P. Bernays, 2nd ed.
[16] D.E. Knuth and P. Bendix, Simple world problems in universal algebras, in: *Computational Problems in Abstract Algebra*, ed. J. Leech (Pergamon Press, 1970) pp. 263–297.
[17] C. Lengauer, On the role of automated theorem proving in the compile-time derivation of concurrency, Journal of Automated Reasoning 1, 1 (1985) 75–101.
[18] Mathlab Group, MACSYMA Reference Manual, Computer Science Laboratory, Massachusetts Institute of Technology, 1977.

[19] J. McCarthy, Applications of Circumscription to formalizing common-sense knowledge, Artificial Intelligence 28, 1 (1986) 89–116.
[20] N.J. Nilsson, *Principles of Artificial Intelligence* (Tioga Publ. Co., 1980).
[21] L.C. Paulson, Lessons learned from LCF: a survey of natural deduction proofs, Comp. J. 28, 5 (1985) 474–479.
[22] W. Polak, *Compiler Specification and Verification*, Lecture Notes in Computer Science 124 (Springer Verlag, 1981).
[23] PRL staff, PRL: Proof Refinement Logic Programmer's Manual, Department of Computer Science, Cornell University, 1984.
[24] R. Reiter, A logic for default reasoning, Artificial Intelligence 13, 1–2 (1980) 81–132.
[25] J.A. Robinson, A machine oriented logic based on the resolution principle, J. ACM 12, 1 (1965) 23–41.
[26] A. Tarski, *A Decision Method for Elementary Algebra and Geometry*, 2nd ed. (University of California Press, 1951).
[27] D.S. Touretzky, *LISP – A Gentle Introduction to Symbolic Computation* (Harper & Row, 1983).
[28] L. Wos, S. Winker and E. Lusk, An automated reasoning system, *AFIPS Conf. Proc. '50*, National Computer Conference, 1981, pp. 697–702.
[29] L. Wos, Solving open questions with an automated theorem-proving program, in: *6th Conf. on Automated Deduction*, ed. D.W. Loveland, Lecture Notes in Computer Science 138 (Springer-Verlag, 1982) pp. 1–31.
[30] L. Wos, R. Overbeek, E. Lusk and J. Boyle, *Automated Reasoning: Introduction and Applications* (Prentice-Hall, 1984).
[31] L. Wos and S. Winker, Open Questions Solved with the Assistance of AURA. In: *Automated Theorem Proving: After 25 Years*, eds. W.W. Bledsoe and D.W. Loveland, Contemporary Mathematics, Vol. 29, American Mathematical Society, 1984, pp. 73–88.
[32] W.-T. Wu, On the decision problem and the mechanization of theorem-proving in elementary geometry, in: *Automated Theorem Proving: After 25 Years*, eds. W.W. Bledsoe and D.W. Loveland, Contemporary Mathematics, Vol. 29, American Mathematical Society, 1984, pp. 213–234.
[33] L.A. Zadeh, A theory of approximate reasoning, in: *Machine Intelligence 9*, eds. J. Hayes, D. Michie and L. Mikulich (Elsevier, 1979) pp. 149–194.