A Survey of Automated Deduction *

Alan Bundy

March 8, 1999

Abstract

We survey research in the automation of deductive inference, from its beginnings in the early history of computing to the present day. We identify and describe the major areas of research interest and their applications. The area is characterised by its wide variety of proof methods, forms of automated deduction and applications.

1 A Potted History of Early Automated Deduction

Deduction is the branch of reasoning formalised by and studied in mathematical logic. The automation of deduction has a long and distinguished history in Artificial Intelligence. For instance, four of the 19 papers in one of the earliest and seminal AI texts, Feigenbaum and Feldman's "Computers and Thought", described implementations of deductive reasoning. These four were:

- a description of a theorem prover for propositional logic, called the Logic Theory Machine, written by Newell, Shaw and Simon, [Newell *et al.*, 1957];
- two papers on Gelernter's Geometry Machine, a theorem prover for Euclidean Geometry, [Gelernter, 1963b; Gelernter, 1963a];
- a description of Slagle's symbolic integrator, Saint. [Slagle, 1963].

As can be seen from these examples, most of the early applications of automated deduction were to proving mathematical theorems. The promise offered to Artificial Intelligence, however, was the automation of commonsense reasoning. This dream is described by McCarthy in his description of the Advice Taker, [McCarthy, 1959], which proposes an application of automated deduction to what we would now call question answering. The earliest implemented applications of automated deduction to commonsense reasoning were to robot plan formation, for instance, the QA series of planners, [Green, 1969], culminating in STRIPS, [Fikes *et al.*, 1971].

The automation of commonsense reasoning was seen to be at the heart of the Artificial Intelligence programme. Sensors, such as television cameras, microphones, keyboards, touch sensors, *etc*, would gather information about the world. This would be used to update a world model. Automated reasoning would then be used to draw inferences from this world model, filling in gaps, answering queries and planning actions. These actions would then be realised by actuators, such as robot arms, speakers, computer screens, mobile vehicles, *etc.*

Automated deduction seemed a very good candidate for the automation of commonsense reasoning. Results from Mathematical Logic suggested that automatic deductive reasoners for first-order, predicate logic could be built that were not only *sound*, *i.e.* only made correct deductions, but were also *complete*, *i.e.* were capable of deducing anything that was true. A theorem of Herbrand's, [Herbrand, 1930], suggested how a sound and complete automatic deduction procedure could be built. Variable-free instances of predicate logic formulae should be systematically

*I would like to thank Richard Boulton, Michael Fisher, Ian Frank, Predrag Janičić, Andrew Ireland and Helen Lowe for feedback on an earlier version of this survey. I would also like to thank Carole Douglas for help in its preparation.

and exhaustively generated then tested for contradiction. Many of the early implementations were based on Herbrand's theorem, *e.g.* [Gilmore, 1960; Davis and Putnam, 1960].

The culmination of this line of work was Robinson's *resolution* method, [Robinson, 1965]. Robinson's key idea was *unification*: an algorithm for finding the most general common instance of two or more logical formulae, if it exists. Unification also generates a substitution, of terms for variables, called a *unifier*. This unifier instantiates the input formulae to their common instance. Using unification, the generation and testing phases of Herbrand's procedure could be merged; contradiction testing was carried out on partially instantiated formulae and guided their further instantiation. Resolution could also be viewed as a process of inferring new formulae from old by matching (using unification) and combining together old formulae. Resolution proved to be much more efficient than earlier techniques and automated deduction researchers switched their attention to it. Many ingenious ways were found to refine resolution, *i.e.* to reduce the generation of new formulae without loss of completeness, thus improving the efficiency of proof search.

Unfortunately, despite this progress in improving efficiency, automatic deduction in the late 1960s and early 1970s was unable to solve hard problems. To prove a mathematical theorem the axioms of the mathematical theory and the negation of the theorem to be proved were first put into a normal form, called *clausal form*. Resolution was then applied exhaustively to the resulting set of *clauses* in the search for a contradiction (the *empty clause*). For non-trivial theorems a very large number of intermediate clauses needed to be generated before the empty clause was found. This generation process exceeded the storage capacity of early computers and the timescales involved were sometimes potentially astronomical. This phenomenon was called *the combinatorial explosion*.

Work on automated deduction was subject to a lot of criticism in the early 1970s. The combinatorial explosion was thought to exclude resolution and similar techniques as viable candidates for the automation of reasoning. Various rival techniques were proposed, for instance, the "procedural representation of knowledge" as embodied in the Micro-Planner AI programming language, [Sussman *et al.*, 1971]. However, on close analysis these alternatives could be seen as offering very similar functionality and performance as existing automated deduction techniques, [Hayes, 1977]. The combinatorial explosion could not be solved by either refinements of resolution or by more *ad hoc* techniques. The solution was seen to lie with the development of *heuristics*, *i.e.* rules of thumb for pruning inference steps (with possible loss of completeness) or for guiding search through the space of inference steps, (see §9).

Another, and more significant, AI criticism of automated deduction was that reasoning is not restricted to deductive inference steps and is not restricted to first-order predicate logic. The reaction to that criticism has been to broaden the coverage of automated reasoning to include a variety of logics, *e.g.* sorted, probabilistic, default, temporal and other modal, *etc.* Some of these logics include non-deductive inference steps, *e.g.* default logics, [Brachman *et al.*, 1991]. This survey, however, will be restricted to deductive inference.

2 Automated Deduction Today

Despite these early criticisms, automated deduction today is in a healthy state. The combination of much faster computers and more efficient coding techniques has enabled resolution theorem provers to prove non-trivial theorems — even open conjectures. A recent notable success has been the proof by the Argonne prover, EQP[1], that Robbins Algebras are boolean algebras, [McCune, 1997]. This was a famous and long standing open conjecture and its automated proof made the front pages of national newspapers all over the world.

Automated deduction has also found significant application in formal methods of system development. Using formal methods computer programs and electronic circuits can be described as logical formulae, as can specifications of their intended behaviour. Questions about such systems can then be posed as automated deduction problems. For instance, automated deduction can be used to ask whether a system meets its specification, to synthesise a system from a specification

[1]A variant of Otter.

or to transform an inefficient system into an equivalent but more efficient one. As commercial pressures force hardware and software producers to provide guarantees of the security and safety of their products, automated deduction is playing an increasingly important role. The CAV conference proceedings give examples of such practical applications, *e.g.* [Alur and Henzinger, 1996].

Automated deduction can be viewed as a form of computation. This has inspired two paradigms for programming language design. These are: logic programming, which is based on the application of resolution to programs expressed as sets of predicate logic clauses (see §11 and [Kowalski, 1979]); and functional programming, which is based on the application of term rewriting to programs expressed as sets of equations in higher-order logic, [Paulson, 1991]. Term rewriting is a rule of inference in which one subterm of a formulae is replaced by an equivalent subterm (see §4). The best known logic programming language is Prolog, [Clocksin and Mellish, 1981], although there are now many variants, *e.g.* a higher-order, constructive version called λProlog, [Miller and Nadathur, 1988] and the various constraint logic programming languages, [Jaffar and Maher, 1994]. Functional programming has been implemented in many languages, *e.g.* Lisp, ML, Miranda, Haskell. There are also hybrid logical and functional languages, *e.g.* LogLisp, Mercury.

Resolution is no longer the dominant automated deduction technique. Formal methods, as well as AI, has forced the field to broaden its coverage. The representation of hardware and software has required the use of higher-order, richly typed, constructive, inductive, modal and linear logics (see §6 and 7). It also puts a strong emphasis on equational reasoning, *e.g.* using term rewriting (see §4). Attempts to emulate commonsense reasoning have required the exploration of default, probabilistic, temporal and other modal logics (see §10). Some of the search problems raised by these logics have defeated total automation, so there has been a lot of interest in interactive theorem proving, in which the burden of finding a proof is divided between the computer and a human user (see §8). Interactive provers usually use more 'human-oriented' presentations of logic, such as sequent calculus, natural deduction or semantic tableaux, rather than the 'machine-oriented' resolution.

The number and size of international conferences to serve the automated deduction community has steadily grown. The Conferences on Automated Deduction (CADEs), which started as a small workshop series in 1974, have now grown into an annual, international series with over a hundred of participants. There are also more specialist conferences devoted to: term rewriting (RTA), semantic tableaux (Tableaux), first-order theorem proving (FTP), higher-order theorem proving (TPHOLs), user interfaces (UITP) and formal verification (CAV). The Journal of Automated Reasoning, which specialises in automated deduction research, was founded in 1985 and is now up to volume 21[2]. The international Association for Automated Reasoning produces a regular newsletter, which is circulated to most automated deduction researchers. Automated deduction research is also reported in many journals and conference proceedings for AI, formal methods and logic and functional programming.

3 Resolution Theorem Proving

In the late 60s and early 70s most work in automated deduction went into finding resolution refinements. Variants of resolution with exotic names: hyper-resolution, model elimination, semantic resolution, RUE, *etc*, abounded, [Loveland, 1978; Wos *et al.*, 1984]. There was also a technique called paramodulation for equational reasoning, which can be viewed as a generalisation of term rewriting, [Robinson and Wos, 1969]. Most of this refinement activity has now died down. The major exception is work on superposition, [Bachmair and Ganzinger, 1994], which seeks to apply work on orderings from termination research[3] to refine resolution.

Most work on resolution now focuses on efficient implementation methods. The goal is to enable resolution-based provers to generate millions of intermediate clauses in a brute force search for a proof, running for hours or days.[4]. Clever indexing techniques are used to enable required

[2] As of August 1998.

[3] The use of orderings to prove the termination of rewriting is discussed in §4.

[4] Even, in extreme cases, years, [Slaney, 1994].

clauses to be retrieved quickly from a database containing millions of candidates, [Overbeek and Lusk, 1980]. Parallelism is exploited to divide the task between many processors, [Lusk *et al.*, 1992]. Technology for precompiling unification and sharing storage is borrowed back from logic programming — so called, Prolog technology theorem proving, [Stickel, 1984]. Algorithms are refined for detecting and removing redundant clauses.

Resolution-based provers are currently evaluated on empirical grounds. Comparisons between provers are mostly based on success rates and run times on standard corpora of problems. The main corpus is the TPTP (Thousands of Problems for Theorem Provers) library, [Sutcliffe *et al.*, 1994]. This is also used as the basis for an annual competition between provers run in conjunction with the CADE conferences, [Suttner and Sutcliffe, 1998]. There is also a lot of interest in tackling open conjectures in mathematics. Conjectures best suited to this approach are combinatorial problems in new areas of mathematics, where human intuitions are less well developed, [Wos, 1993]. The Robbins Algebra conjecture is a good example. Brute force search through all the various possibilities is often an appropriate approach to such problems — and machines are much better suited to brute force solutions than humans.

Resolution is a refutation-based procedure; it proves a conjecture when it derives a contradiction in the form of the empty clause. When the conjecture is false the search for a contradiction may terminate with failure or the search may continue unsuccessfully forever. In the case of terminating failure a counterexample can be constructed from the finite search space. This potential can be exploited to build models of formulae from unsuccessful attempts to find proofs of their negations. There has been a lot of interest in exploiting this potential recently; several model constructing programs have been built and successful applications developed, [Slaney, 1994]. There is even a category in the CADE theorem prover competition for such model constructing programs.

4 Term Rewriting

A common pattern of reasoning in mathematics is to replace a subterm in an expression by an equivalent subterm. Some proofs consist of a chain of such replacements in which one expression is gradually transformed into another. For instance, when the conjecture is an equation the left-hand side may be transformed into the right-hand side. This transformation procedure is formalised in automated deduction as *term rewriting* using *rewrite rules*, [Huet and Oppen, 1980]. A rewrite rule is an equation that has been oriented in one direction: conventionally left to right. To *rewrite* an expression, first a subterm of the expression is identified for replacement. Then a rewrite rule is selected whose left-hand side matches this subterm using a one-side unification. The rewrite rule is instantiated so that its left-hand side is identical to the subterm. The subterm is then replaced with the instantiated right-hand side of the rule.

Of course, this procedure is incomplete in general, *i.e.* when a conjecture can be proved using some of the rewrite rules in both orientations, but not with a single orientation. Sometimes, however, rewriting *is* complete, for instance, if a set of rewrite rules can be shown to be *confluent*. A set of rewrite rules is confluent if, whenever an expression can be rewritten into two distinct expressions then these two can both be further rewritten into a third common expression. A simple inductive proof shows that whenever there is a proof using a confluent set of rewrite rules as equations in both orientations then there is a rewriting proof.

If, in addition, the set of rewrite rules is terminating then rewriting constitutes a decision procedure, *i.e.* it is terminate with either a proof or a refutation of an equational conjecture . A set of rewrite rules is terminating if there are no infinite chains of rewriting using them. To prove an equation, for instance, each side of the equation is rewritten until no more rewriting is possible, which will happen after a finite number of steps. This final rewritten expression is called a *normal form*. The equation is true if and only if the two normal forms are identical. Confluence also ensures that these normal forms are unique, so there is no need to search.

In 1970 a breakthrough occurred in term rewriting with the discovery by Knuth and Bendix of a procedure for testing a set of rewrite rules for confluence, [Knuth and Bendix, 1970]. The Knuth-Bendix test compares the left-hand sides of rewrite rules to see whether they could give rise to a choice of rewriting. So called *critical pairs* are constructed and then tested to see if they rewrite to an identical normal form. If all critical pairs rewrite to identical normal forms then the set of rewrite rules is confluent. Moreover, Knuth and Bendix proposed a procedure to transform a non-confluent set of rules into a confluent one. If a critical pair generates two non-identical (but necessarily equal) normal forms then these are formed into the left- and right-hand side of a rewrite rule and added to the set of rewrite rules. This new set may, in turn, generate new critical pairs (using the new rule) and this whole process may not terminate. But if it does, then the final set of rewrite rules will be confluent. The invention of this procedure triggered a huge activity in term rewriting, which is now a major sub-area of automated deduction with its own conference series (RTA).

This has also spurred a renewed interest in proving termination of sets of rewrite rules. Termination proving is equivalent to the halting problem, so only partial solutions are possible. However, current techniques are capable of proving the termination of many of the sets of rewrite rules that arise in practice, *cf* [Walther, 1994b], for instance. These techniques can also be automated, so that theorem provers can establish the termination of rewrite rule sets before applying them. All the techniques use a measure which maps expressions into a well-founded set. A *well-founded set* is an ordered set of objects in which there are no infinite descending chains. For each rewrite rule the measure of its left-hand side is shown to be greater than that of its right-hand side under this order. The ordering must also be shown to be preserved by the rewriting process, *i.e.* by substitution and replacement of subterms. Then each rewriting strictly reduces the measure of the rewritten expression. Since there are no infinite descending chains of measures this reduction cannot continue indefinitely, so rewriting must terminate.

Simple measures use the natural numbers as the well-founded set. More sophisticated measures use polynomials over natural numbers or combine previous measures using pairs and multi-sets (*aka* bags). One of the most interesting developments is the use of *term orderings*, in which the expressions themselves form the well-founded set, with the order being defined by a set of syntactic inference rules. The best known of these term orderings is *recursive path ordering* (rpo), [Dershowitz, 1982].

As mentioned above, term rewriting can be viewed as a special case of paramodulation. Paramodulation extends rewriting in three ways. Firstly, equations can potentially be used in both orientations. Secondly, two-way unification rather than one-way matching is used to instantiate the replaced subterms as well as the equations. Thirdly, the equations are disjoined with other formulae to form a full clause. This relationship between paramodulation and rewriting has inspired new developments in both areas.

Rewriting has been extended towards paramodulation in two directions. Firstly, conditions have been added to rewrite rules. Usually these conditions must be established before the rule is applied. Secondly, two-way unification can be used to instantiate variables in the subterm. This form of rewriting is called *narrowing*, [Fay, 1979; Hanus, 1994]. It is useful for proving existential theorems, with the existential variables in the conjecture being represented by free variables and instantiated during the narrowing proof.

Paramodulation has been refined by using term orderings from termination theory to restrict its application without loss of completeness, [Bachmair and Ganzinger, 1990]. This has in turn inspired the work on superposition mentioned in §3, in which similar ideas are applied to resolution.

Most applications of term rewriting apply the rewrite rules *exhaustively*, *i.e.* until the expression being rewritten is in normal form. Recently, there has been interest in *selective* rewriting, in which restrictions are imposed on rewriting. The best known of these selective rewriting techniques is *rippling*, [Bundy *et al.*, 1993; Basin and Walsh, 1996]. Meta-level annotations are inserted into the expressions to be rewritten and into the rewrite rules. The effect of these annotations is to prevent some sub-expressions from being rewritten and to impose restrictions on the way in which other sub-expressions can be rewritten. The motive behind these restrictions is to direct the rewriting to produce an expression which matches some hypothesis. The main application is to the step cases of inductive proofs, in which the induction conclusion is directed towards a match with the induction hypothesis. It has also been used: to find closed forms to sums, [Walsh *et al.*, 1992]; to solve limit theorems, [Yoshida *et al.*, 1994]; and to prove equalities, [Hutter, 1997].

5 Built-in Unification

Robinson's original unification was based only on the syntactic structure of the formulae to be unified. So if two formulae had instances which were equal, but not syntactically identical, then Robinson's algorithm would declare then non-unifiable. In 1972, Plotkin built the associativity axiom into the unification algorithm, [Plotkin, 1972]. Associative unification finds a substitution which instantiates the input formulae to instances which are equal modulo associativity, but not necessarily identical. When using associative unification the associative axiom need not be used explicitly in the proof, *i.e.* it is completely built-into the unification algorithm.

This work initiated a major sub-area of automated deduction in which different combinations of axioms are built-into the unification algorithm, *e.g.* associativity, commutativity, idempotency, distributivity, *etc* and combinations of these. Built-in unification is especially valuable where an axiom causes problems if included in a set of clauses or rewrite rules. For instance, the commutative law causes non-termination when used as a rewrite rule, but commutative unification is quite efficient. So rewriting using commutative unification is an attractive solution. Many functions are both associative and commutative, so building both of these into unification (AC-unification) has received a lot of attention. A good survey of built-in unification algorithms can be found in [Jouannaud and Kirchner, 1991].

Given unifiable formulae, Robinson's unification algorithm generates a unique, most-general unifier. "Most-general" means that any other unifier is an instance of it. Built-in unification problems are not all so well behaved. For some there is more than one unifier – sometimes infinitely many. Nor does a most-general unifier always exist. Some unification problems are even undecidable. Unification problems can be classified along these various dimensions: unique/finite/infinite number of unifiers; existence of most-general unifiers; decidable/undecidable.

Unification algorithms are nowadays presented as a set of transformation rules, which rewrite input formulae into substitutions, [Jouannaud and Kirchner, 1991]. Such presentations facilitate proofs that the algorithms are both sound and complete and inform calculations of their complexity. They also enable general-purpose unification algorithms to be designed, *i.e.* algorithms which take a set of axioms to be built-in as an additional input. The work on general-purpose unification algorithms derives from work on term rewriting (see §4), since the axioms must usually be represented as a confluent rewrite rule set.

One of the major achievements of built-in unification was Huet's higher-order unification algorithm, [Huet, 1975; Jouannaud and Kirchner, 1991], which builds-in the α, β and optionally the η rules of λcalculus. This algorithm makes automated higher-order theorem proving possible. Higher-order unification is a badly behaved problem: there can be infinitely many unifiers and the problem is undecidable. Research into higher-order unification is active in at least two directions: the extension of the algorithm to new kinds of higher-order logic, such as constructive type theory; and the search for decidable, but still useful, sub-cases of the problem, which are, therefore, better behaved. For instance, if the input formulae are restricted to, so called, *higher-order patterns* then unique, most-general unifiers exist, [Miller, 1991].

Unification finds the most-general common instance of two formulae. The dual problem, *anti-unification*, finds the least-general common generalisation, *i.e.* a formula which has both of the input formulae as instances. Plotkin invented anti-unification in 1969, [Plotkin, 1969], but interest in it has recently revived in the area of machine learning, where it is used to find the general form of two or more examples of some concept. For instance, in *Inductive Logic Programming*, [Muggleton, 1991], it is used to learn a general logic program from instances of it.

6 Higher-Order Logic and Type Theory

Many problems in both mathematics and formal methods are more naturally represented in a higher-order logic than in first-order. For instance, functional programming languages are usually based on typed λ-calculus, which is a higher-order logic, [Barendregt, 1985]. So the reasoning about functional programs is naturally done in higher-order logic. Reasoning about limited resources can be done in linear logic, [Girard *et al.*, 1995], which models resources with assumptions and places restrictions on the number of times they can be used in proofs. Similarly, mathematics uses second and higher-order functions, such as summation, differentiation, integration, *etc*, which are more naturally reasoned about in a higher-order framework.

First-order automatic deduction techniques are readily adapted to higher-order deduction by replacing first-order unification with higher-order unification. This is not quite as straightforward as it sounds. For instance, the potentially infinite branching of higher-order unification needs to be factored into the search space, *e.g.* by allowing backtracking to return alternative unifiers. Three examples of higher-order theorem provers are TPS [Andrews *et al.*, 1996], HOL [Gordon, 1988] and PVS [Owre *et al.*, 1992].

Some applications of theorem proving require even richer logics. For instance, automated deduction can be used to synthesise programs (and circuits) meeting a specification of their intended behaviour. A conjecture is posed that for any inputs an output exists obeying some specified relationship with the input. For instance, if a sorting program is required the output might be specified as an ordered permutation of the input. The required program can be extracted from a proof of the conjecture; different proofs yielding different programs. However, this synthesis can fail to yield a program if the conjecture is proved in a "pure existence" proof, namely a proof which shows the existence of an output without showing how it can be constructed from the inputs. This problem can be avoided by proving the conjecture in a *constructive* logic, *i.e.* one from which pure existence proofs are excluded.

Formal methods proof obligations (see §2) are also best conducted in a logic with a rich type structure, *i.e.* one in which the various data-structures (*e.g.* integers, reals, arrays, lists, trees, *etc*) and the arities of the procedures (*e.g.* lists to integers) are represented as types. Rich type structures, higher-order logic and constructive proofs are combined in *constructive type theories*, [Martin-Lof, 1970]. There are now a number of theorem provers based on constructive type theories, *e.g.* Coq [Dowek *et al.*, 1991], NUPRL [Constable *et al.*, 1986], LEGO [Luo and Pollack, 1992], ALF, [Augustsson *et al.*, 1990]. Due to the difficulty of automating interesting proofs in some of these logics, most of these theorem provers are interactive (see §8).

A wide variety of different logics have been developed for formal methods. These include many different: constructive type theories, temporal logics for reasoning about changing behaviour over time, process algebras for reasoning about concurrent programs, dynamic logics for reasoning about imperative programs, *etc.* Building theorem provers for each of these logics is a massive challenge, especially since the logic design is itself often a variable in the research programme, so that the theorem prover is under constant modification. One answer to this is to build generic theorem provers, which take a specification of the logic as an input. the theorem prover is then easily reconfigured. One of the most popular of these is Paulson's Isabelle, [Paulson, 1986]. Constructive type theories also turn out to be well suited as meta-logics for the specification of the input logic. They also have technical advantages, like providing a generic unification algorithm. Generic theorem provers taking this approach are usually called *logical frameworks*, [Huet and Plotkin, 1991].

7 Inductive Theorem Proving

In mathematics, formal methods and common-sense reasoning we often want to reason about repetition. Repetition might arise from: recursively defined mathematical objects, data-structures or procedures; iteration in programs; feedback loops in circuits; behaviour over time; or general object descriptions with a parameter. This repetition is often unbounded. To reason about unbounded repetition it is usually essential to use a rule of mathematical induction. Such rules are used to reduce a universally quantified conjecture into some number of base and step cases. In the base cases the conjecture is proved for some initial values, *e.g.* the number 0. In the step cases the theorem is assumed for a generic value, *e.g.* n and, using this assumption, proved for some subsequent value, *e.g.* $n + 1$. In this way the conjecture is incrementally proved for an infinite succession of values.

Logical theories which include induction rules are subject to some negative theoretical results which cause problems for automation.

1. They are usually *incomplete*, *i.e.* they contain true but unprovable formulae, [Gödel, 1931]. This manifests itself in requiring an unlimited number of distinct induction rules.

2. They do not admit *cut elimination*, which means that arbitrary intermediate formulae may need to be proved and then used to prove the current conjecture, [Kreisel, 1965]. This manifests itself in requiring the generalisation of conjectures and/or the introduction of intermediate lemmas. In contrast, resolution generates any necessary intermediate formulae as a side effect.

Both of these negative results introduce potentially infinite branching points into the search space. At any stage: an unbounded number of induction rules can be specially constructed and applied; the current subgoal can be generalised in an unbounded number of ways; or any formula can be introduced as an intermediate lemma. Special heuristics are required to control these branching points. Some use the failure of initial, restricted proof attempts to introduce patches which extend the search space. Failures in rippling (see §4) have proved especially fruitful in suggesting such proof patches, [Ireland and Bundy, 1996].

Two main approaches have been developed for automating inductive proof: explicit and implicit. In explicit induction additional inductive rules of inference are added to the logic and used in the search for a proof, [Walther, 1994a]. Since there are potentially infinitely many induction rules, it is usual to have a method for dynamically creating new rules customised to the current conjecture, [Walther, 1993]. It is also necessary to have a *cut rule* for generalising conjectures and creating intermediate lemmas. Explicit induction theorem provers include: Nqthm [Boyer and Moore, 1988], INKA [Biundo *et al.*, 1986], RRL [Kapur and Zhang, 1995] and Oyster/CLaM [Bundy *et al.*, 1990].

A conjecture is an inductive consequence of a theory if and only if it is consistent to add it to that theory. Implicit induction is based on this theorem. This method is also called *inductionless induction* or *inductive completion*. Consistency is usually tested by trying to express the extended theory as a confluent and terminating set of rewrite rules and then rewriting the conjecture to normal form, [Kapur *et al.*, 1991]. Although explicit and implicit induction sound very different, close analysis of the rewriting process in implicit induction reveals proof steps which are very similar to the base and step cases of explicit induction. In fact, implicit induction can be viewed as a form of explicit induction using a term order, such as recursive path order, as the induction order, [Reddy, 1990]. Implicit induction provers include: RRL [Kapur and Zhang, 1995] (which uses both implicit and explicit induction) and SPIKE [Bouhoula *et al.*, 1992].

8 Interactive Theorem Proving

The state of the art of automated deduction is that theorem proving programs will often fail to prove non-trivial theorems. One solution to this is to develop *interactive* theorem provers, where the task is divided between the computer and a human user. The role of the user can vary from specifying each proof step to setting some global parameters for an otherwise automated run. Usually, the human role is to make the difficult proof guidance decisions while the computer takes care of the book-keeping and the routine steps. All proof systems lie on a continuum between fully automatic and fully interactive; most 'automatic' systems have some facility for user interaction and most 'interactive' systems have some degree of automation. So although this is a survey of *automated* deduction, interaction must be discussed.

Many interactive provers provide some kind of macro facility so that users can apply multiple proof steps with one command. Of these, the best developed macro facility is called *tactics*, [Gordon *et al.*, 1979]. A tactic is a computer program which directs a theorem prover to apply some rules of inference. A primitive tactic applies one rule of inference. Tactics are composed together, using *tacticals*, into sequences, conditional cases, iterations, *etc.* They are typically used for: simplifying expressions by putting them into normal form; applying decision procedures; and following other common patterns of reasoning. Tactics are organised hierarchically with big tactics defined in terms of smaller ones and ultimately in terms of rules of inference. There are many tactic-based provers, *e.g.* HOL [Gordon, 1988], Nuprl [Constable *et al.*, 1986], Isabelle [Paulson, 1986].

In order to be able to guide an interactive prover it is vital for a human user to interpret the structure of the emerging proof. One way to achieve this is for the interactive prover to, use a logic which presents proofs in a 'natural' format, such as Gentzen's sequent calculus, natural deduction or semantic tableaux. Resolution, with its clausal form and powerful single inference rule is usually considered too 'machine-oriented' for interactive provers[5].

Interactive provers frequently have elaborate graphical user interfaces to try to present emerging proofs and the operations on them in a congenial and accessible way. For instance, some systems present the overall structure of a proof as a tree in which the nodes represent formulae and the branches represent proof steps connecting them. These trees can be organised hierarchically, with nodes representing big tactics unpacking into sub-trees representing their smaller sub-tactics. This can be used to show the overall structure of the proof succinctly with the detail being revealed on demand. Menus, buttons, mouse clicks, *etc* can be used to present the various options to the user in a user-friendly way. For instance, the "proof by pointing" style selects rules of inference by clicking the mouse on appropriate parts of the conjecture. As well as guiding the proof, the user may need access to libraries of theories, conjectures, definitions, previous lemmas, *etc.* They may want to: store and recover partial proofs; reset global options; load or delete definitions, lemmas, heuristics, *etc.* Providing all the functionality that users may want, while orienting them in the partial proof and not overwhelming them with too much information is a very hard and unsolved problem. The annual workshops on User Interfaces for Theorem Provers (UITP) is a good source of research in this area, [Backhouse, 1998].

9 Meta-Reasoning and Proof Methods

Human mathematicians do not find proofs by combining low level rules of inference. They adopt a higher-level of abstraction combining common patterns of proof using meta-reasoning. To emulate their success it is necessary to automate such higher-level reasoning. Tactics are one route to do this by providing powerful, high-level reasoning steps for the prover. They are used to encode proof methods in a meta-language, like ML [Paulson, 1991].

One common pattern of reasoning in proofs is the application of a decision procedure to solve a decidable subproblem. Popular decision procedures are available for:

Propositional logic: There has been a lot of recent interest in using tautology checkers to verify digital electronic circuits which can be modelled as propositional formulae. Very efficient tautology checkers have been built, of which the best known are based on Ordered Binary Decision Diagrams (OBDDs), [Bryant, 1992]. However, tautology checking is an NP-complete problem so, failing a favourable solution to $P = NP$, even the most efficient tautology checkers are exponential in the worse case.

Presburger arithmetic: The additive fragments of integer and real number arithmetic is decidable, [Presburger, 1930]. Sub-goals in this fragment often occur when reasoning about iteration in program verification. Various more efficient variants of Presburger's original algorithm have been developed, [Cooper, 1972; Hodes, 1971], although they are all super-exponential in the worse case. Much work has limited to the quantifier-free sub-case, since this is all that is required for verification proofs and has much lower complexity, [Bledsoe, 1974].

Euclidian geometry: Decision procedures for decidable subsets of geometry can be obtained by translating geometric problems into algebraic ones and then using algebraic decision

[5] Although, the logic programming work has shown that resolution proofs can be quite 'natural'.

procedures such as those for Presburger arithmetic, [Wu, 1997; Chou, 1988]. These are available for a range of geometries.

There are also techniques for combining together decision procedures for disjoint decidable theories, [Nelson and Oppen, 1979; Shostak, 1984].

Many decision procedures are embedded in computer algebra systems. There is a lot of interest in interfacing theorem provers to computer algebra systems to be able to access these procedures. There is also interest in the other direction in using theorem provers to verify decision procedures or for checking their preconditions. A special issue of JAR addresses both of these interactions, [Kapur and Wos, 1998].

Many proof methods of use in automated deduction systems are for non-decidable fragments of theories. Even though these are not decision procedures, they can have a high heuristic success rate and so be of practical use. One such family of useful, heuristic methods are for difference reduction, *i.e.* they identify differences between two formulae to be shown equal and then systematically reduce these differences. One of the best known such methods is *rippling* (see §4).

Meta-reasoning is used to reason about (object-level) logic theories in a meta-theory. The domain of discourse of the meta-theory is the formulae and proof methods of the object-level theory. For instance, meta-reasoning might be used to analyse the current goal, choose an appropriate method of proof and apply it. This lifts the search space from the object-theory to the meta-theory, which is often better behaved, *i.e.* has a smaller search space. One example of meta-reasoning is *reflection*, in which theorems in a meta-theory are related, via reflection rules, to theorems in an object-theory, and *vice versa*, [Weyhrauch, 1980]. Another example is *proof planning*, in which meta-reasoning is used to build a global outline of a proof, which is then used to guide the detailed proof, [Bundy, 1991]. A third example is *analogy*, in which an old proof is used as a plan to guide the proof of a new theorem, [Owen, 1990]. Fourthly, many decision procedures reason at the meta-level. Meta-reasoning can also be used to explain the high-level structure of proofs, *e.g.* to the user of an interactive prover, [Lowe and Duncan, 1997].

10 Commonsense Reasoning

The everyday reasoning of humans involves knowledge, belief, time, uncertainty and guessing based on sparse information. A variety of 'non-classical' logics have been developed to capture these 'commonsense' aspects of reasoning. Modal logics contain special operators for representing beliefs of agents and time, [Chellas, 1980]. Uncertainty logics associate degrees of certainty with logical formulae, [Pearl, 1988]. Default logics have rules which infer *defeasible* steps, in that conclusions which can be assumed but later withdrawn in the face of contradictory evidence, [Brachman *et al.*, 1991]. Commonsense reasoning can be automated by building automatic theorem provers for these logics.

From an automated proof viewpoint most of these non-classical logics are much worse behaved than classical, first-order, predicate logic, *i.e.* they generate an even bigger combinatorial explosion. A common approach to solving this problem is to restrict the logic to reduce the amount of search. For instance, lots of work on modal logics is restricted to the propositional fragment, *i.e.* to propositional logic with modal operators. Many of these propositional, modal logics are decidable with relatively efficient decision procedures, [Halpern and Moses, 1992].

Another popular approach to automating reasoning in non-classical logics is to *reify* them into classical logics. This is done by formalising the semantics of the non-classical logic in a classical logic, [Wallen, 1990; Ohlbach, 1991]. For instance, the semantics of modal logics is expressed in a system of linked 'possible worlds'. To formalise this semantics each predicate of the classical logic is given an additional argument which specifies a possible world in which it is asserted. Universal and existential quantification over these possible worlds then represents the modal operators. Links between the possible worlds are expressed as relations between them, which can often be embedded into the unification algorithm. In this way modal reasoning can be automated (for instance) via a conventional resolution prover.

11 Logic Programming & Deductive Databases

Prolog gives a procedural interpretation to a fragment of first order, predicate logic. It applies resolution to *Horn clauses*, which are disjunctions of negated propositions and at most one unnegated proposition. This fragment has a natural procedural interpretation, [Kowalski, 1979], but if two unnegated propositions are included in a clause this interpretation begins to break down. There is significant interest in trying to give a procedural interpretation to more expressive logics. This is most readily done, not in classical predicate logic, but in constructive logics. One of the best developed examples is Miller's λProlog logic programming language, [Miller and Nadathur, 1988], which is based on a higher-order, constructive logic. Other researchers have given a logic programming interpretation to constructive type theory [Pfenning, 1991], linear logic and temporal logic..

Since logic programs are logical formulae they seem especially well suited to the application of formal methods. Unfortunately, practical logic programming languages, like Prolog, often contain non-logical features for the sake of efficiency, practicality and to support meta-programming. Prolog, for instance, has: an unsound unification algorithm[6]; predicates for asserting and retracting clauses; predicates for syntactic analysis; and the 'cut' for cutting out part of the search space. Research is directed at developing semantics of logic programming languages which capture some of the non-logical features, [Borger and Rosenzweig, 1994]. These semantics are then used to reason about the operation of logic programs. For instance, we might want to transform Prolog programs into programs with the same run time behaviour (except, perhaps for their speed) and not just into programs which are logically equivalent but, for instance, explore the search space in a different order.

There is also research at developing logic programming languages with a more declarative semantics. The challenge is to provide comparable power and efficiency to Prolog within a purely logical language. For instance, meta-programming facilities can be provided with an explicit and cleanly separated meta-level. One of the best known attempts to do this is the Gödel logic programming language, [Hill and Lloyd, 1994].

One method for improving the efficiency of logic programs is *partial evaluation*, [Komorowski, 1982]. Suppose a program is to be run successively on similar input. The computation performed on each run might have large parts in common, leading to redundancy if it is re-performed each time. This can be prevented by performing the common computation once and saving it in the form of a transformed version of the original program. This can be implemented by running the original program on a generalised form of the inputs which captures their similarities and using the result as the transformed program.

Running logic programs on generalised or abstract data can also be used as an analysis technique called *abstract interpretation*. The abstraction might, for instance, throw away all details of the data except for its type or its mode[7]. Running a logic program on this abstract data can be used to infer its type or mode signature, [Mellish, 1987].

One of the most successful extensions to logic programming is to combine it with constraint reasoning, called *constraint logic programming*, [Jaffar and Maher, 1994]. Formulae are divided between those to be treated by logic programming and those to be treated as constraints. The latter are typically equations, inequalities, *etc.* The constraints are solved by a decision procedure, *e.g.* an equation or inequality solver or optimiser, such as the Simplex algorithm. Constraint logic programming has found many industrial applications. It enables the smooth combination of qualitative and quantitative methods of problem solving. It allows traditional operational research methods to be augmented with symbolic reasoning techniques.

Logic programming languages provide a form of default reasoning by interpreting the failure to prove a goal as evidence for its negation, so called *negation as failure*. There is research to relate this technique to other forms of default reasoning, *e.g.* non-monotonic logics, circumscription, *etc.* There is also interest in relating this default technique to the use of integrity constraints in databases, [Kowalski *et al.*, 1987].

[6]Missing the 'occurs check'.

[7]The mode of a logic program specifies which arguments can be input, output or either.

In fact, logic programs can be seen as a logical extension of relational databases, in which rules can be used to derive new data not explicitly stored in the original database. Datalog is the best known of the logic programming languages adapted for use as extended databases, [Ceri *et al.*, 1990]. It is purely relational, *i.e.* it differs from Prolog, say, by having no functions, but extends relational databases with the use of rules.

12 Conclusion

Automated deduction has grown into a broad field in which a wide variety of proof methods is used on a wide variety of logics and applied to a wide variety of applications. Proof methods range from interactive to automatic — and include every stage in-between. Both machine-oriented methods, such as decision procedures and resolution, and human-oriented methods, such as natural deduction and rewriting, are used and mixed. The logics range from classical first-order to constructive type theory and take in temporal, other modal, default, uncertainty, *etc* on the way. All these are applied to: proof obligations in formal methods; the design and implementation of programming languages; commonsense reasoning in artificial intelligence; relational databases; knowledge-based systems; *etc.* The system descriptions in the CADE proceedings contain a good record of the rich variety of implemented automated deduction provers.

Automated deduction techniques are being integrated with other techniques into practical systems. For instance, the Amphion system, [Lowry *et al.*, 1994], uses a theorem prover for synthesising programs to control space probes from a library of subroutines. The theorem prover is hidden from the user and a graphical front-end enables people unfamiliar with automatic deduction or logic to use it. Hybrid approaches are being developed. For instance, the Stanford temporal prover, [Manna, 1994] combines automatic deduction with model checking.

Throughout its development automated deduction has drawn on and fed back to work in mathematics and especially logic. Most of the logics have come from mathematical or philosophical logic. However, some logics, such as default and uncertainty logics, arose from automated deduction and then attracted mathematical interest. The interests of automated deduction have also refocussed mathematical interest, *cf.* the rising mathematical interest in constructive logics because of their computational applications. Mathematicians have also been interested to solve some of the mathematical problems that arise from automated deduction. These problems include: finding termination proofs; proving soundness, completeness and complexity results for proof methods; inventing new decision procedures.

There is no end to the variety of problems, methods and applications thrown up by automated deduction. The amount of work being conducted is greater than ever in its history, but the number of open research problems has grown, not diminished. We are entering an exciting phase of research as new application areas are coming into maturity and exciting new research directions are being identified.

References

[Alur and Henzinger, 1996] Rajeev Alur and Thomas A. Henzinger, editors. *Proceedings of the 1996 Conference on Computer-Aided Verification*, number 1102 in LNCS, New Brunswick, New Jersey, U. S. A., 1996. Springer-Verlag.

[Andrews *et al.*, 1996] P. B. Andrews, M. Bishop, I. Sunil, D. Nesmith, F. Pfenning, and H. Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

[Augustsson *et al.*, 1990] L. Augustsson, T. Coquand, and B. Nordström. A short description of another logical framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.

[Bachmair and Ganzinger, 1990] L. Bachmair and H. Ganzinger. On restrictions of ordered paramodulation with simplification. In M. E. Stickel, editor, *Proc. 10th Int. Conf. on Automated Deduction, Kaiserslautern*, volume 449, pages 427–441. Springer-Verlag, 1990.

[Bachmair and Ganzinger, 1994] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994. Revised version of Research Report MPI-I-91-208, 1991.

[Backhouse, 1998] R. Backhouse, editor. *2nd International Workshop on User Interfaces for Theorem Provers*, 1998.

[Barendregt, 1985] H. P. Barendregt. *The Lambda Calculus.* Elsevier, 1985.

[Basin and Walsh, 1996] David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[Biundo *et al.*, 1986] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe induction theorem proving system. In Joerg Siekmann, editor, *8th International Conference on Automated Deduction*, pages 672–674. Springer-Verlag, 1986. Springer Lecture Notes in Computer Science No. 230.

[Bledsoe, 1974] W. W. Bledsoe. The Sup-Inf method in Presburger arithmetic. Memo ATP-18, Department of Mathematics, University of Texas at Austin, USA, Dec 1974.

[Borger and Rosenzweig, 1994] E. Borger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1994.

[Bouhoula *et al.*, 1992] A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE: an automatic theorem prover. In *Proceedings of LPAR '92*, number 624 in LNAI. Springer-Verlag, July 1992.

[Boyer and Moore, 1988] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Perspectives in Computing, Vol 23.

[Brachman *et al.*, 1991] R. Brachman, H. Levesque, and Reiter. R, editors. *Artificial Intelligence*, volume 49. Elsevier, 1991.

[Bryant, 1992] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Bundy *et al.*, 1990] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al.*, 1993] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Bundy, 1991] Alan Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991. Also available from Edinburgh as DAI Research Paper 445.

[Bundy, 1994] Alan Bundy, editor. *12th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, Nancy, France, 1994. Springer-Verlag.

[Ceri *et al.*, 1990] Stefano Ceri, Georg Gottlob, and Leitzia Tanca. *Logic Programming and Databases.* Surveys in Computer Science. Springer-Verlag, Berlin, 1990.

[Chellas, 1980] B.F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, 1980.

[Chou, 1988] S-C Chou. *Mechanical Geometry Theorem Proving.* Reidel Pub. Co., Dordrecht, 1988.

[Clocksin and Mellish, 1981] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer Verlag, 1981.

[Constable *et al.*, 1986] R. L. Constable, S. F. Allen, H. M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Cooper, 1972] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. Edinburgh University Press, 1972.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Association for Computing Machinery*, 7:201–215, 1960.

[Dershowitz, 1982] Nachum Dershowitz. Orderings for term rewriting systems. *Journal of Theoretical Computer Science*, 17(3):279–301, 1982.

[Dowek *et al.*, 1991] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, and B. Werner. The Coq proof assistant user's guide, version 5.6. Technical Report 134, INRIA, 1991.

[Fay, 1979] M.J. Fay. First-order unification in an equational theory. In *Procs. of the Fourth Workshop on Automated Deduction*, pages 161–167. Academic Press, 1979.

[Fikes *et al.*, 1971] R. E. Fikes, , and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Gelernter, 1963a] H. Gelernter. Empirical explorations of the geometry theorem-proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 153–163. McGraw Hill, 1963.

[Gelernter, 1963b] H. Gelernter. Realization of a geometry theorem-proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 134–152. McGraw Hill, 1963.

[Gilmore, 1960] P. C. Gilmore. A proof method for quantificational theory. *IBM J Res. Dev.*, 4:28–35, 1960.

[Girard *et al.*, 1995] J.-Y. Girard, Y. Lafont, and L. Regnier, editors. *Advances in Linear Logic.* Number 222 in London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, 1995.

[Gödel, 1931] K. Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 38:173–198, 1931. English translation in [Heijenoort, 1967].

[Gordon *et al.*, 1979] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[Gordon, 1988] M. J. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1988.

[Green, 1969] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–239, Washington, D. C., U. S. A., 1969.

[Halpern and Moses, 1992] J. Y. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379, 1992.

[Hanus, 1994] M. Hanus. The integration of functions into logic programming: from theory to practice. *J. Logic Programming*, 19 & 20:583–628, 1994.

[Hayes, 1977] P Hayes. In defence of logic. In *Proceedings of IJCAI-77*, pages 559–565. International Joint Conference on Artificial Intelligence, 1977.

[Heijenoort, 1967] J van Heijenoort. *From Frege to Gödel: a source book in Mathematical Logic, 1879-1931.* Harvard University Press, Cambridge, Mass, 1967.

[Herbrand, 1930] J. Herbrand. Researches in the theory of demonstration. In J van Heijenoort, editor, *From Frege to Goedel: a source book in Mathematical Logic, 1879-1931*, pages 525–581. Harvard University Press, Cambridge, Mass, 1930.

[Hill and Lloyd, 1994] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[Hodes, 1971] Louis Hodes. Solving problems by formula manipulation in logic and linear inequalities. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 553–559, Imperial College, London, England, 1971. The British Computer Society.

[Huet and Oppen, 1980] G. Huet and D. C. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal languages: Perspectives and open problems.* Academic Press, 1980. Presented at the conference on formal language theory, Santa Barbara, 1979. Available from SRI International as technical report CSL-111.

[Huet and Plotkin, 1991] G. Huet and G. D. Plotkin. *Logical Frameworks.* CUP, 1991.

[Huet, 1975] G. Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hutter, 1997] D. Hutter. Coloring terms to control equational reasoning. *Journal of Automated Reasoning*, 18(3):399–442, 1997.

[Ireland and Bundy, 1996] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[Jaffar and Maher, 1994] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[Jouannaud and Kirchner, 1991] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*, chapter 8, pages 257–321. MIT Press, 1991.

[Kapur and Wos, 1998] D. Kapur and l. Wos, editors. *Journal of Automated Reasoning*, volume 21. Kluwer, 1998.

[Kapur and Zhang, 1995] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. of Computer Mathematics with Applications*, 29(2):91–114, 1995.

[Kapur *et al.*, 1991] D. Kapur, P. Narendran, and H. Zhang. Automating inductionless induction by test sets. *Journal of Symbolic Computation*, 11:83–111, 1991.

[Knuth and Bendix, 1970] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[Komorowski, 1982] H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of PROLOG. In *Proceedings of the ninth conference on the Principles of Programming Languages (POPL)*, pages 225–267, Albuquerque, New Mexico, 1982. ACM.

[Kowalski *et al.*, 1987] R. Kowalski, F. Sadri, and P. Soper. Integrity checking in deductive databases. In *Proc. 13th VLDB, Brighton*, pages 61–69. Morgan Kaufmann, 1987.

[Kowalski, 1979] R. Kowalski. *Logic for Problem Solving*. Artificial Intelligence Series. North Holland, 1979.

[Kreisel, 1965] G. Kreisel. Mathematical logic. In T. Saaty, editor, *Lectures on Modern Mathematics*, volume 3, pages 95–195. J. Wiley & Sons, 1965.

[Loveland, 1978] D. W. Loveland. *Automated theorem proving: A logical basis*, volume 6 of *Fundamental studies in Computer Science*. North Holland, 1978.

[Lowe and Duncan, 1997] H. Lowe and D. Duncan. XBarnacle: Making theorem provers more accessible. In William McCune, editor, *14th International Conference on Automated Deduction*, pages 404–408. Springer-Verlag, 1997.

[Lowry *et al.*, 1994] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*, Charlotte, North Carolina, October 1994.

[Luo and Pollack, 1992] Z. Luo and R. Pollack. Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, May 1992.

[Lusk *et al.*, 1992] E.L. Lusk, W.W. McCune, and J. Slaney. Roo: A parallel theorem prover. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, pages 731–734, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[Manna, 1994] Z. et al Manna. Step: the stanford temporal prover. Technical Report STAN-CS-TR;94-1518, Stanford University, 1994.

[Martin-Lof, 1970] P. Martin-Lof. *Notes on Constructive Mathematics*. Almqvist and Wiksell, Stockholm, 1970.

[McCarthy, 1959] J. McCarthy. Programs with common sense. In *Mechanisation of Thought Processes (Proceedings of a symposium held at the National Physics Laboratory, London, Nov 1959)*, pages 77–84, London, 1959. HMSO.

[McCune, 1997] W. McCune. Solution of the Robbins problem. *J. Automated Reasoning*, 19(3):263–276, 1997.

[Mellish, 1987] C. S. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–197. Ellis Horwood, 1987.

[Miller and Nadathur, 1988] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming*. MIT Press, 1988.

[Miller, 1991] D. Miller. Unification of simply typed lambda-terms as logic programming. In P.K. Furukawa, editor, *Proc.1991 Joint Int. Conf. Logic Programming*, pages 253–281. MIT Press, 1991.

[Muggleton, 1991] S.H. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[Nelson and Oppen, 1979] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[Newell *et al.*, 1957] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations with the Logic Theory Machine. In *Proc. West. Joint Comp. Conf.*, pages 218–239, 1957. Reproduced in Computers and Thought (eds Feigenbaum and Feldman), McGraw Hill, New York, pp 109-133, 1963.

[Ohlbach, 1991] H.J. Ohlbach. Semantics based translation methods for modal logics. *Journal of Logic and Computation*, 1(5):691–746, 1991.

[Overbeek and Lusk, 1980] R. A. Overbeek and E. L. Lusk. Logic data structures and control architecture for implementation of theorem proving programs. In W. Bibel and R. Kowalski, editors, *5th International Conference on Automated Deduction*. Springer Verlag, 1980. Lecture Notes in Computer Science No. 87.

[Owen, 1990] S. Owen. *Analogy for Automated Reasoning*. Academic Press Ltd, 1990.

[Owre *et al.*, 1992] S. Owre, J. M. Rushby, and N. Shankar. PVS : An integrated approach to specification and verification. Tech report, SRI International, 1992.

[Paulson, 1986] L.C. Paulson. Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Paulson, 1991] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Pearl, 1988] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[Pfenning, 1991] F. Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 182. Cambridge University Press, 1991.

[Plotkin, 1969] G. Plotkin. A note on inductive generalization. In D Michie and B Meltzer, editors, *Machine Intelligence 5*, pages 153–164. Edinburgh University Press, 1969.

[Plotkin, 1972] G. Plotkin. Building-in equational theories. In D Michie and B Meltzer, editors, *Machine Intelligence 7*, pages 73–90. Edinburgh University Press, 1972.

[Presburger, 1930] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929*, pages 92–101, 395. Warsaw, 1930. Annotated English version also available [Stansifer, 1984].

[Reddy, 1990] U. S. Reddy. Term rewriting induction. In *Proc. of Tenth International Conference on Automated Deduction*. Springer-Verlag, 1990.

[Robinson and Wos, 1969] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In D. Michie, editor, *Machine Intelligence 4*, pages 103–33. Edinburgh University Press, 1969.

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

[Shostak, 1984] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984. Also: *Proceedings of the 6th International Conference on Automated Deduction*, volume 138 of *Lecture Notes in Computer Science*, pages 209–222. Springer-Verlag, June 1982.

[Slagle, 1963] J. R. Slagle. A heuristic program that solves symbolic integration problems in freshman calculus. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 191–203. McGraw Hill, 1963.

[Slaney, 1994] J. Slaney. The crisis in finite mathematics: automated reasoning as cause and cure. In Bundy [1994], pages 1–13.

[Stansifer, 1984] Ryan Stansifer. Presburger's article on integer arithmetic: Remarks and translation. Technical Report TR 84-639, Department of Computer Science, Cornell University, September 1984.

[Stickel, 1984] M. E. Stickel. A Prolog technology theorem prover. *New Generation Computing*, 2(4):371–83, 1984.

[Sussman *et al.*, 1971] G.J. Sussman, T. Winograd, and E. Charniak. Micro-planner reference manual. Technical Report 203a, MIT AI Lab, 1971.

[Sutcliffe *et al.*, 1994] G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In Bundy [1994], pages 252–266.

[Suttner and Sutcliffe, 1998] C. Suttner and G. Sutcliffe. The CADE-14 ATP system competition. *Journal of Automated Reasoning*, 21(1):99–134, 1998.

[Wallen, 1990] L. A. Wallen. *Automated Proof Search in Non-Classical Logics*. MIT Press, London, 1990.

[Walsh *et al.*, 1992] T. Walsh, A. Nunes, and Alan Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th International Conference on Automated Deduction*, pages 325–339. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.

[Walther, 1993] C. Walther. Combining induction axioms by machine. In *Proceedings of IJCAI-93*, pages 95–101. International Joint Conference on Artificial Intelligence, 1993.

[Walther, 1994a] C. Walther. Mathematical induction. In C. J. Hogger D. M. Gabbay and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 12, pages 122–227. Oxford University Press, Oxford, 1994.

[Walther, 1994b] C. Walther. On proving termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[Weyhrauch, 1980] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

[Wos *et al.*, 1984] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, 1984.

[Wos, 1993] L. Wos. Automated reasoning answers open questions. *Notices of the AMS*, 5(1):15–26, January 1993.

[Wu, 1997] Wen-Tsün Wu. The Char-Set method and its applications to automated reasoning. In William McCune, editor, *14th International Conference on Automated Deduction*, pages 1–4. Springer-Verlag, 1997.

[Yoshida *et al.*, 1994] Tetsuya Yoshida, Alan Bundy, Ian Green, Toby Walsh, and David Basin. Coloured rippling: An extension of a theorem proving heuristic. In A. G. Cohn, editor, *In proceedings of ECAI-94*, pages 85–89. John Wiley, 1994.