Automation of Proof by Mathematical Induction

Alan Bundy	Robert S. Boyer	Deepak Kapur	Christoph Walther
University of Edinburgh	University of Texas	SUNY Albany	TH Darmstadt

Mathematical induction is required for reasoning about objects or events containing repetition, e.g. computer programs with recursion or iteration, electronic circuits with feedback loops or parameterized components. Thus mathematical induction is a key enabling technology for the use of formal methods in information technology. Failure to automate inductive reasoning is one of the major obstacles to the widespread use of formal methods in industrial hardware and software development.

Recent developments in automatic theorem proving promise significant improvements in our ability to automate mathematical induction. As a result of these developments, the functionality of inductive theorem provers has begun to improve. Moreover, there are some promising signs that even more significant improvements are possible. This enlarges the applicability of automated induction for "real world" problems and research topics for application have been discussed on the seminar.

Automated induction is a relatively small subfield of automated reasoning. Research is based on two competing paradigms each having its merits but also its shortcomings as compared with the other:

- Implicit induction evolved from Knuth-Bendix-Completion and most of the work based on this paradigm was performed by researchers concerned with term rewriting systems in general.
- Explicit induction has its roots in traditional automated theorem proving. It resembles the more familiar idea of theorem proving by induction where induction axioms are explicitly given and specific inference techniques are tailored for proving base and step formulas.

This seminar brought together leading scientists from both areas to discuss recent advancements within both paradigms, to evaluate and compare the state of the art and to work for a synthesis of both approaches. It summarized the results of a series of workshops held on automated induction in conjunction with the CADE conferences 1992 (Saratoga Springs) and 1994 (Nancy) and the AAAI conference 1993 (Washington DC).

The success of this meeting was due in no small part to the Dagstuhl Seminar Center and its staff for creating such a friendly and productive environment. The organizers and participants greatly appreciate their effort. The organizers also thank Jürgen Giesl and Martin Protzen for their support in many organizational details.

Contents

Dagstuhl Seminar 9530
Automation of Proof by Mathematical Induction
24. 7. – 28. 7. 1995

Monday, July 24, 1995

Time			*Speaker/Chair*	*Title*
9:00	–	9:10	*The Organizers*	Welcome to Dagstuhl
			David Musser	
9:10	–	9:35	Mahadevan Subramania	Reasoning about Hardware Descriptions in RRL
9:40	–	10:05	Wolfgang Küchlin	Hardware-Verification by Term-Rewriting Induction
10:10	–	10:20	Discussion	
10:20	–	10:50	Break	
			Christoph Walther	
10:50	–	11:15	Dieter Hutter	Using C-Terms in Automating Induction
11:20	–	11:45	Alan Bundy & Vincent Lombart	Relational Rippling: A General Approach
11:50	–	12:00	Discussion	
12:00	–	14:30	Lunch	
			Deepak Kapur	
14:30	–	14:55	David Musser	Automated Verification of Generic Algorithms in the C++ Standard Template Library using Subgoal Induction
15:00	–	15:25	Leo Bertossi	Proving Database Integrity Constraints by Mathematical Induction
15:30	–	15:40	Discussion	
15:40	–	16:10	Break	
			David Basin	
16:10	–	16:35	Jürgen Giesl	Proving Termination of Algorithms Automatically
16:40	–	17:05	Jürgen Brauburger	Conditional Termination
17:10	–	17:20	Discussion	
18:00			Dinner	

Dagstuhl Seminar 9530
Automation of Proof by Mathematical Induction
24. 7. – 28. 7. 1995

Tuesday, July 25, 1995

Time			*Speaker/Chair*	*Title*
			Robert S. Boyer	
9:10	–	9:35	Peter Dybjer	Type Theory for Dummies
9:40	–	10:05	Herman Geuvers	Calculus of Constructions with Congruence Types
10:10	–	10:20	Discussion	
10:20	–	10:50	Break	
			Andrew Ireland	
10:50	–	11:15	Martin Protzen	Synthesis and Manipulation of Proof Predicates
11:20	–	11:45	Claus Sengler	Induction on Non-Freely Generated Data Types
11:50	–	12:00	Discussion	
12:00	–	14:30	Lunch	
			Michaël Rusinowitch	
14:30	–	14:55	Deepak Kapur	Failure Analysis of Induction Schemes and their Role in Generalization
15:00	–	15:25	Andrew Ireland	The Problem of Generalization in the Context of Mathematical Induction
15:30	–	15:40	Discussion	
15:40	–	16:10	Break	
			Dieter Hutter	
16:10	–	16:35	Francisco J. Cantu	The Use of Inductive Proof Plans for Hardware Verification
16:40	–	17:05	Laurence Pierre	Application of Mathematical Induction to the Formal Proof of Hardware and Embedding in a CAD Framework
17:10	–	17:20	Discussion	
18:00			Dinner	
20:00			Panel	"When (if ever) will Inductive Theorem Proving be of Economic Significance?"

Dagstuhl Seminar 9530
Automation of Proof by Mathematical Induction
24. 7. – 28. 7. 1995

Wednesday, July 26, 1995

Time			*Speaker/Chair*	*Title*
			Alan Bundy	
9:10	–	9:35	Natarajan Shankar	High-Level Strategies for Induction Proofs in PVS
9:40	–	10:05	Stefan Gerberding	Incremental Tactical Theorem Proving
10:10	–	10:20	Discussion	
10:20	–	10:50	Break	
			David McAllester	
10:50	–	11:15	Sentot Kromodimoeljo	Certifying Induction in the Never Theorem Prover
11:20	–	11:45	Bernhard Gramlich	On Evaluation Criteria for Inductive Theorem Proving Systems
11:50	–	12:00	Discussion	
12:00	–	13:45	Lunch	
14:00	–	23.00	Excursion	
18:00			Dinner	

Dagstuhl Seminar 9530
Automation of Proof by Mathematical Induction
24. 7. – 28. 7. 1995

Thursday, July 27, 1995

Time			*Speaker/Chair*	*Title*
			Toby Walsh	
9:10	–	9:35	David McAllester	Grammar Rewriting in Inductive Proofs
9:40	–	10:05	Deborah Weber–Wulff	Problems with Parsing Proofs in NQTHM
10:10	–	10:20	Discussion	
10:20	–	10:50	Break	
			Natarajan Shankar	
10:50	–	11:15	Toby Walsh	A Calculus for Rippling
11:20	–	11:45	Thomas Kolbe & Christoph Walther	Reusing Proofs
11:50	–	12:00	Discussion	
12:00	–	14:30	Lunch	
			Wolfgang Küchlin	
14:30	–	14:55	Jürgen Avenhaus	Theorem Proving in Hierarchical Conditional Specifications
15:00	–	15:25	Ulrich Kühler	Inference Rules for Induction Theorem Proving in Hierarchical Specifications
15:30	–	15:40	Discussion	
15:40	–	16:10	Break	
			Jürgen Avenhaus	
16:10	–	16:35	David Basin	(Avoiding) Induction using Monadic Second-Order Logic
16:40	–	17:05	Narjes Berregeb	Extending Test Set Induction Proofs in AC Theories
17:10	–	17:20	Discussion	
18:00			Dinner	

Dagstuhl Seminar 9530
Automation of Proof by Mathematical Induction
24. 7. – 28. 7. 1995

Friday, July 28, 1995

Time			*Speaker/Chair*	*Title*
9:10	–	10:20	*Robert S. Boyer* —	Open Session
10:20	–	10:50	Break	
10:50	–	12:00	*Alan Bundy* —	Challenge Problems for Automated Induction and Evaluation Criteria for Inductive Theorem Proving Systems
12:00	–	14:30	Lunch	

Theorem Proving in Hierarchical Specifications

Jürgen Avenhaus Klaus Madlener
FB Informatik, Universität Kaiserslautern

In the talk we consider clausal specifications for describing programs on the level of functional design. The axioms of such a specification $spec = (sig, E, \mathcal{A})$ are positive/negative conditional equations, they define new operators on top of a fixed built-in algebra $\mathcal{A}$. The new operators may only be partially defined by E. We define the semantics of *spec* to be the quotient algebra $\mathcal{A}_{spec}$ of the free term algebra according to the congruence relation given by E and $\mathcal{A}$. In this approach partiality is modelled by using order sorted specifications and order sorted algebras: For each sort we introduce a super sort for the junk terms. We prove $\mathcal{A}_{spec}$ to be initial in the class of all models of *spec*, provided *spec* is a consistent extension of $\mathcal{A}$. So, a clause is called an inductive theorem of *spec* iff it holds in $\mathcal{A}_{spec}$. We present an inference system to prove inductive validity of a set of clauses. This inference system may also be used to disprove the inductive validity of a clause.

(Avoiding) Induction using Monadic Second Order Logics

David Basin
MPI-Saarbrücken

We show how the second-order monadic theory of strings can be used to specify hardware components and their behavior. This logic admits a decision procedure and counter-model generator based on canonical automata for formulas. We have used a system implementing these concepts to verify, or find errors in, a number of circuits proposed in the literature. The techniques we use make it easier to identify regularity in circuits, including those that are parameterized or have parameterized behavioral specifications. Our proofs are semantic and do not require lemmas or induction as would be needed when employing a conventional theory of strings as a recursive data type.

Bibliography

[Bas95] David A. Basin and Nils Klarlund. Hardware Verification using Monadic Second-Order Logic CAV 1995.

Extending SPIKE to associative and commutative theories

Narjes Berregeb
(joint work with Michaël Rusinowitch and Adel Bouhoula)
INRIA & CRIN Lorraine

Many mathematical theories like groups, unitary rings or boolean algebra, involve associative and commutative (AC) functions. These operators are hard to handle for automatic deduction and generate complex proofs.

So, in order to perform more efficient proofs, we extended the prover by test set induction, SPIKE [Bou95], to AC conditional theories. An important advantage of such extension is that we only need AC matching, and we do not use AC unification like in other inductive completion methods [JK89] [Bün91]. We also improved and integrated some simplification techniques. The system obtained is correct and refutationally complete under some reasonable hypotheses. Experiments have shown the gain in efficiency we obtain: proofs are more natural and require less human interaction. In particular, we proved the correctness of two digital circuits, a rippled carry adder (in one run) and an iterated additions multiplier (with one lemma).

Bibliography

[JK89] J-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. In *Information and computation* , 82:1-33, 1989

[Bou95] A. Bouhoula. Preuves automatiques par recurrence dans les theóries conditionnelles. PhD thesis, Université de Nancy I, 1994.

[Bün91] R. Bündgen. Term completion versus Algebraic completion. PhD thesis, University of Tübingen, 1991.

Proving Database Integrity Constraints by Automated Induction

Leo Bertossi (Catholic Univ. Chile), Deepak Kapur (SUNY, Albany)
Javier Pinto (Catholic Univ. Chile), Pablo Saez (Catholic Univ. Chile)
Mahadevan Subramaniam (SUNY, Albany)

We are concerned with incorporating automation of proofs of integrity constraints into SCDBR, a reasoner on database updates specifications that is under construction at the Catholic University of Chile.

The specifications handled by the system are given according to Ray Reiter's formalism for specifying dynamic structures that change by the effects of executed actions. That formalism gives a simple solution to the frame problem in the situation calculus.

Integrity constraints are properties that must hold at every legal state in the evolution of a (relational) database. The "legality" property of states is defined by induction.

In this paper, we show how to use RRL (Kapur & Zhang), a mechanical theorem prover based on conditional term rewriting, for proving integrity constraints by induction. RRL's cover set method for generating appropriate induction schemas has turned out to be very powerful for that purpose.

We show how to specify first–order quantifiers in RRL by means of the so–called "bounded quantification", and also how to generate and prove lemmas that have to do with "the logic of bounded quantification".

Finally, we present some techniques for proving the logical lemmas, and discuss the role of "generalization" in their proofs.

Conditional Termination

Jürgen Brauburger
TH Darmstadt

We introduce *termination predicates* which are sufficient for the termination of *partial recursively defined algorithms* and we develop an automated *synthesis* procedure for termination predicates. Termination predicates – which are *totally defined* by algorithms – enable us to control the use of partial functions, since non-termination can be recognized in advance. In *induction theorem proving* we can use termination predicates to formulate and to verify statements containing partial functions. Furthermore, termination predicates can be used to show *termination of imperative programs* according to the following approach: An imperative program is transformed into a functional one, where loops often result in partial functions. Then for each partial function a termination predicate is synthesized which can be used for the termination proof of the functional program.
Starting from termination proofs for total functions we collect requirements which must be satisfied by termination predicates: A termination predicate of a recursively defined algorithm f must imply, that (1) for each recursive call in f the parameters are decreasing wrt. a well-founded order, and (2) that each function call occuring in the algorithm terminates. Based on this characterization we develop a procedure which synthesizes a termination predicate for a given recursively defined algorithm fully automatically. Our generation method needs no search and it is independent of any particular well-founded order.

Relational Rippling: A General Approach

Alan Bundy
University of Edinburgh

Vincent Lombart
University of Edinburgh

We propose a new version of rippling, called *relational rippling*. Rippling is a heuristic for guiding proof search, especially in the step cases of inductive proofs. Relational rippling is designed for representations in which value passing is by shared existential variables, as opposed to function nesting. Thus relational rippling can be used to guide reasoning about logic programs or circuits represented as relations.

In rippling, annotations are placed on formulae to indicate which bits must remain unchanged during rewriting (the *skeleton*) and which bits must be moved (the *wave-fronts*) and in what direction. Relational rippling requires additional annotations in order to define the skeleton and to impose a direction for the wave-fronts to move in. With these annotations it is possible to prove termination of relational rippling.

Inductive Proof Plans for Hardware Verification

Francisco J. Cantú
University of Edinburgh AI Department

We describe the use of inductive proof plans for verifying combinational and sequential hardware. In verifying combinational hardware typically we show that the specification equals the implementation. If either the specification or the implementation contain recursive components then the proof goes by induction.

In verifying sequential synchronous hardware we assume that if the specification at time t equals the implementation at time l, then the specification at $t+1$ equals the implementation at time l' where t is a abstract time scale corresponding to the specification, l is more detailed time scale corresponding to the implementation and l' corresponds to $t+1$ in this scale.

In verifying both types of hardware the *Rippling* heuristic plays an important role in guiding proofs by induction and in doing proofs where difference match/unification is required.

We have been able to verify a set of combinational devices which include n-bit adders, subtracters, dividers, multipliers, exponentiators, factorials, arithmetic logic units, and shifter units, and sequential units such as an n-bit counter and a small microprocessor (the Gordon computer, in progress).

The development time for these verifications range between 3 days to 3 weeks, depending upon the complexity of the device and the run time typically varies between 30 seconds to 6 minutes.

Type Theory for Dummies
or
Type Theory for Functional Programmers

Peter Dybjer
Chalmers Technical University, Göteborg

In this talk I give a brief description of Martin-Löf's Intuitionistic Type Theory (ITT) as a modification of an idealized standard functional programming language. The core of such an idealized functional language is the simply typed lambda calculus. The user can then extend this core with new recursive datatypes and new functions defined by general recursion. The core of ITT is instead the dependent type lambda calculus. Similarly, this core can be extended by the user with new (possibly dependent) recursive datatypes and new recursive functions. In order that all (typable) programs in the language terminate one only allows datatypes the elements of which can be viewed as well-founded trees and that functions are defined by primitive (structural) recursion on the inductive generation of the elements of a datatype.

Using different words one can also say that ITT is an intuitionistic theory of iterated inductive definitions in a framework of dependent types, which integrates an intuitionistic logic via the Curry-Howard identification of propositions and types (and proofs and programs). This theory is very expressive and even intended to serve as a full-scale framework for predicative constructive mathematics.

Incremental Tactical Theorem Proving

Stefan Gerberding
TH Darmstadt

We propose a new approach to *fully automated* tactical theorem proving — *incremental tactical theorem proving* — which is superior to the conventional proof planning or heuristics driven approach to automated tactical theorem proving. Incremental tactical theorem proving enhances the capabilities of proof planning, because more information about the proof and the proof plan is available. Domain knowledge about proof planing (strategy information) is encoded *declaratively* in the meta-rules, therefore the proof planning process is no longer executed by an uninformed planner. This leads to more powerful proof plans.

The meta-rule interpreter interleaves proof planning (strategy) steps with plan "executing" (tactic) steps. Thereby many problems concerning the failure of tactics to achieve the desired results become obsolete. Thus we avoid replanning. Because our meta-rule interpreter knows about the history of the (partial) proof and the (partial) proof plan by being able to access the (partial) proof-tree the system is able to synthesize more successful proof plans.

The declarative representation of strategy knowledge in the meta-rules eases the maintenance and development of the meta-reasoning component, since the information about proof planning resp. proof strategy is no longer implicitly contained in the proof planner. Furthermore, because knowledge about tactics and about strategy is stored together in the system of meta-rules tuning the tactics wrt. the strategy or vice versa is easier.

Another advantage of incremental tactical theorem proving is the simplification of failure recovery for tactics and replanning. Due to the incremental planning the divergence of the plan execution and the expected results is avoided, since the differences between expected results and the results of actually executing a tactic cannot accumulate. Thereby also the quality of the proof plan is increased, because the next step is planned on the basis of more reliable information about the current goal. In conventional tactical theorem proving this information is extrapolated by the proof planner using the postcondition-slots of the methods.

We argue that incremental tactical theorem proving is more intuitive wrt. human problem solving capabilities. When a human mathematician proves a theorem, he starts working from an idea how to proceed. These ideas are not as specific as proof plans but they represent more general knowledge about the domain. The ideas are represented by our meta-rule. Using his idea the human applies his methods (tactics) step-by-step. The human uses the intermediate results to decide the next step. In other words, the human mathematician is also interleaving object-level reasoning and meta-reasoning.

We are currently working on the implementation of a prototype incremental tactical theorem prover which is based on the ITPS inka. Future plans include the development of an explanation component to support the maintenance of the system of meta-rules, the implementation of an optimizer for the evaluation of preconditions, and we plan to investigate mixed forward / backward reasoning.

The Calculus of Constructions with Congruence Types

Herman Geuvers
Technological University Eindhoven, NL

Gilles Barthe
University of Nijmegen, NL

The combination of type systems with algebraic rewriting systems has given rise to algebraico-functional languages, a class of very powerful programming languages. Yet these frameworks only allow for a limited interaction between the algebraic rewriting systems and the type theory. For example, if $\mathbb{Z}$ is defined as an algebraic type, one cannot define the absolute value or prove that every integer is either positive or negative. This serious objection to algebraico-functional languages is in fact due to the absence of induction principles for algebraic types and so one might be tempted to formulate such principles. However, the task is not so easy if confluence of the system has to be preserved. Indeed, the computations attached to induction principles and those attached to algebraic types do not interact satisfactorily.

To solve these complications we opt for a two-level approach, in which every algebraic type is accompanied with the inductive type of its signature and related to it by suitable axioms for quotients. For the case of $\mathbb{Z}$, this amounts to having an inductive type $\underline{Z}$ with constructors $\underline{0}, \underline{s}$ and $\underline{p}$ (the type of terms of the signature of $\mathbb{Z}$) and an algebraic type Z with constants $0 : Z$, $s : Z \to Z$ and $p : Z \to Z$ and rewrite rules $p(sx) \to_\rho x$ and $s(px) \to_\rho x$. A 'class' map $[-]$ and a 'representant' map rep take care of the interaction between the types $\underline{Z}$ and Z: if $t : \underline{Z}$, then $[t] : Z$ and if $t : Z$, then $\mathsf{rep}\, t : \underline{Z}$. To ensure that Z and $\underline{Z}$ behave satisfactorily one must add equalities between terms. This is done by adding rewrite rules (the χ-rules), which are, for the case of $\mathbb{Z}$, as follows.

- $[\underline{0}] \to_\chi 0$, $[\underline{s}x] \to_\chi s[x]$ and $[\underline{p}x] \to_\chi p[x]$, stating that $[-]$ is the unique map from the (initial) algebra $\underline{Z}$ to the algebra Z.
- $\mathsf{rep}\ 0 \to_\chi \underline{0}$, $\mathsf{rep}\ (st) \to_\chi \underline{s}\ (\mathsf{rep}\ t)$ and $\mathsf{rep}\ (pt) \to_\chi \underline{p}\ (\mathsf{rep}\ t)$, for t a closed term in normal form. Together with the previous rules this states that Z is universal as a quotient of $\underline{Z}$ and the rewrite relation. (We think of rep as a map which assigns to every equivalence class a canonical representative.)
- $[\mathsf{rep}\ x] \to_\chi x$, stating that $[-]$ is a surjective map from $\underline{Z}$ to Z.

Furthermore, we have to ensure that for q and t of type $\underline{Z}$, $[q]$ and $[t]$ are equal iff (q, t) is in the smallest equivalence relation on $\underline{Z}$ that contains the rewrite rules. This is done axiomatically, where we take for the equality the Leibiniz-equality.

In this set-up, one can transfer the induction principle (on $\underline{Z}$) to the algebraic type (Z) without affecting the confluence of the system. We claim that such a formalism, which we call *congruence types*, is suited for giving a faithful representation of canonical term-rewriting systems both from the logical and the computational point of view.

We see three important uses of congruence types.

1 Represent types (such as $\mathbb{Z}$) that can not be defined as inductive types, because they arise as a quotient of an inductive type.

2 Obtain a better computational behavior of a definable function on an inductive type. This is achieved by defining an inductive type with 'extra' constructors and adding rewrite rules to ensure that the extra constructor represents the function we have in mind.

3 Use the algebra of terms Σ (the inductive type) to prove properties of the quotient structure $\mathcal{S}(= \Sigma/R)$. In this case one really uses the inductive reasoning (about the algebra of terms Σ) to derive a statement about the quotient type $\mathcal{S}$. An example is the case where one has defined an interpretation $[\![-]\!]^A$ from the term-algebra Σ to a structure A, such that $[\![-]\!]^A$ is compatible with the rewrite rules R. In the formalism of congruence types, one can then prove that $\forall x, y{:}\Sigma.[x] =_{\mathcal{S}} [y] \to [\![x]\!]^A =_A [\![y]\!]^A$. The gain here is that if t and t' are terms of Σ such that $[t]$ and $[t']$ are convertible, then the premise $[t] =_{\mathcal{S}} [t']$ is immediate, and hence the conclusion $[\![t]\!]^A =_A [\![t']\!]^A$ follows.

Proving Termination of Algorithms Automatically

Jürgen Giesl
TH Darmstadt

When proving statements about algorithmically specified functions by induction, *termination* of the algorithms has to be verified because of two reasons.

- The algorithms are translated into axioms for subsequent deductions. *Termination* (and determinism) of the algorithms is sufficient for the *consistency* of these axioms.
- Every terminating algorithm suggests an induction scheme. *Termination* of the algorithm guarantees the *soundness* of the resulting induction axioms.

We have developed a method for automated termination proofs of *recursively defined algorithms* [Gie95b], [Gie95c]. For every algorithm a well-founded ordering relating inputs and corresponding arguments of recursive calls has to be generated.

For termination proofs of *term rewriting systems* several approaches to synthesize suited well-founded *term orderings* have been suggested. But unfortunately term orderings cannot be directly used for termination proofs of *algorithms*, because term orderings may conflict with the algorithmic definitions of *defined* function symbols.

A straightforward solution is the restriction to term orderings which respect the *semantics* of the called algorithms. Another obvious solution is to transform the algorithms into a term rewriting system and to prove the system's termination instead. But with both of these solutions most termination proofs will not succeed with any of the commonly used term orderings, i.e. these solutions result in a *too weak* termination criterion.

Therefore we have developed a calculus to transform inequalities between inputs and arguments of recursive calls into inequalities *without defined function symbols*. This transformation is an *abduction*, i.e. the resulting inequalities define a relation which *contains* the relation defined by the original inequalities as a subset. Consequently well-foundedness of this new relation is sufficient for the termination of the algorithm.

As the resulting inequalities contain no algorithmically defined function symbols, they define a relation whose well-foundedness can be directly proved with term orderings. For this purpose we use a procedure for the generation of *polynomial orderings* [Gie95a].

Hence our transformation enables the application of *term orderings* for termination proofs of *algorithms*. This results in a more powerful technique than previous procedures for automated termination proofs of algorithms. Our method has been implemented within the induction theorem proving system INKA.

Bibliography

[Gie95a] J. Giesl. Generating Polynomial Orderings for Termination Proofs. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, Kaiserslautern, Germany, 1995.

[Gie95b] J. Giesl. Automated Termination Proofs with Measure Functions. In *Proceedings of the 19th Annual German Conference on Artificial Intelligence*, Bielefeld, Germany, 1995.

[Gie95c] J. Giesl. Termination Analysis for Functional Programs using Term Orderings. In *Proceedings of the Second International Static Analysis Symposium*, Glasgow, Scotland, 1995.

On Evaluation Criteria for Inductive Theorem Proving Systems

Bernhard Gramlich
Universität Kaiserslautern

We discuss the problem of how to define and how to measure progress in inductive theorem proving (ITP) technology. To this end we first identify some crucial ingredients and aspects of ITP systems that have to be taken into account when trying to develop reasonable evaluation criteria for such systems. In particular, we elaborate on essential aspects of the underlying design philosophy of ITP systems and on (the need of) a conceptual model of the entire *proof engineering process* which is of great importance when tackling non-trivial proof tasks. Then we exhibit various reasons why defining and measuring progress in ITP technology seems to be rather difficult. In fact, we do not provide a fully worked out approach for systematic evaluation criteria, but rather give an outline of what should be taken into account when doing so, and why. Finally, we discuss the related problem of constructing in a systematic way collections of interesting examples for ITP systems.

Using Colored Terms Everywhere

Dieter Hutter
German Research Center for Artificial Intelligence, Saarbrücken

Heuristics for judging similarities between formulas and subsequently reducing differences have been applied to automated deduction since the 1950s, when Newell, Shaw, and Simon built their first "logic machine". Since the later 60s, a similar theme of difference identification and reduction appears in the field of resolution theorem proving.

In the field of inductive theorem proving syntactical differences between the induction hypothesis and induction conclusion are used in order to guide the proof [Bundy93], or [Hutter90, Hutter91]. This method to guide induction proofs is called rippling / colouring terms. First, the syntactical differences between induction hypothesis and induction conclusion are shaded. A shaded area is called a *wave-front* while the other parts belong to the *skeleton*. Analogously, syntactical differences between both sides of equations or implications given in the database are shaded. These formulas are classified depending on the locations of the wave-fronts inside the unshaded expressions (e.g. wave-fronts on both sides, wave-fronts only on the right-hand side, or wave-fronts only on the left-hand side). Using these annotated equations we are able to move, insert, or delete wave-fronts within the conclusion. This *rippling* of wave-fronts allows to reduce the differences between conclusion and hypothesis in a goal directed way. Rippling involves very little search and always terminates since wave-fronts are only moved in some well-founded way.

Based on the success of rippling in the field of inductive theorem proving, several attempts have been made to extend the scope of rippling to general equational reasoning. In [BaWa93] procedures are presented to determine the syntactical differences of arbitrary terms in the notion of skeletons and wave-fronts. Based on this method, in [ClHu94] heuristics are developed to guide the proof process with the help of syntactical differences.

This talks presents an extension of the colouring method to higher-order logics. Thus our work provides a formal basis to the implementation of rippling in a higher-order setting which is required e.g. in case of middle-out reasoning. But the set of possible applications of our method is not limited to automated deduction. From an abstract point of view, the colouring technique allows adding annotations to symbol occurrences in λ-terms. Thus in contrast to other semantic annotation techniques like sorts, it is possible to encode syntactic information and use that to guide inferencing processes. This makes it plausible to expect applications (of generalizations) of our technique in computational linguistics and natural language semantics, where the use of higher-order unification is used in the context of Montegovian semantics.

Bibliography

[BaWa93] David Basin and Toby Walsh. Difference Unification. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (ICJAI)* ed. Ruzena Bajcs, p. 116-122, Morgan Kaufmann, Chambery, France, 1993

[Bundy93] Alan Bundy and Andrew Stevens and Frank van Harmelen and Andrew Ireland and Alan Smaill, Rippling: a heuristic for guiding inductive proofs, *Journal of Artificial Intelligence, p. 185-253, vol. 62, 1993*

[ClHu94] Cleve, J. and Hutter, D. A methodology for equational reasoning. In *Proceedings Hawaii International Conference on System Sciences 27 Volume III: Information Systems : DSS/Knowledge-based Systems* Eds : Jay F. Nunamaker, jr. and Ralph H. Sprague, jr., p. 569 - 578, IEEE Computer Society Press, Los Alamitos, California, 1994

[Hutter90] Dieter Hutter. Guiding Induction Proofs. In *Proceedings of the 10th Conference on Automated Deduction*, Ed. Mark Stickel, LNCS 449, Springer, Kaiserslautern, Germany, 1990

[Hutter91] Dieter Hutter, Mustergesteuerte Strategien zum Beweisen von Gleichheiten, Ph.D. Thesis, Universität Karlsruhe, 1991

[HuKo95] Dieter Hutter and Michael Kohlhase. A colored version of the lambda-calculus. Technical report, University of Saarland, 1995

On Guiding the Generalization of Conjectures

Andrew Ireland and Alan Bundy
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge, Edinburgh
Scotland, UK
EH1 1HN

It is known that the cut-elimination theorem does not hold for inductive theories. Generalization is underpinned by the cut-rule of inference and consequently introduces an infinite branching point into the search space for inductive proofs. For this reason generalization is a major obstacle to the automation of proof by mathematical induction. A technique, or *proof critic*, for guiding generalization is presented. Our proof critic builds upon *rippling*, a heuristic based *proof method* for guiding inductive proof. The critic enables us to exploit failed proof attempts in the search for generalizations. In particular we focus upon generalization through the introduction of accumulator variables.

Failure Analysis of induction schemes and role of generalization

Deepak Kapur
SUNY at Albany

M. Subramaniam
SUNY at Albany

An approach for predicting failure of induction schemes used in an inductive proof of a conjecture is developed. The approach is based both on *local analysis* as well as *global analysis* of definitions and lemmas. In local analysis, *unchangeable positions* of a function definition, an idea due to Boyer and Moore, is exploited. Rules preserving unchangeable positions are defined and used to predict failure. Local analysis serves as a basis for classifying induction schemes as flawed and unflawed. The notion of flawed schemes a la Boyer and Moore is extended by taking into account, the context provided by the conjecture to the schemes. We are thus able to classify as flawed, schemes which result in unsuccessful proof attempts but which would be declared unflawed by Boyer and Moore's heuristics and hence considered for proof attempts. In global analysis, right sides of definitions are examined to analyze functions that could possibly appear as an argument to another function. This information is used to determine whether the use of a particular induction scheme for an induction proof could lead to a new function f not in the conjecture, appearing as an argument to another function g such that no rule can eliminate an occurrence of f. Local and global analyses are used to develop conditions under which it is not possible to generate an induction proof based on a particular induction scheme, and in avoiding proof attempts which would surely lead to failure. In case all induction schemes possible in a conjecture are declared flawed by local and global analyses, the generalization heuristic can be used to get a more general version of the conjecture which leads to at least one unflawed scheme. Such a controlled use of generalization also avoids unsuccessful proof attempts. Additional lemmas that are not preserving unchangeable positions can be used as a dynamic context to propose new cover sets leading to different unchangeable positions of functions which may result in unflawed schemes. Examples of properties of functions on numbers and lists are used to illustrate various concepts.

Reusing Proofs

Thomas Kolbe
TH Darmstadt

Christoph Walther
TH Darmstadt

We investigate the improvement of induction theorem provers by reusing previously computed proofs. Our approach for reuse is based on generalizing computed proofs by replacing function symbols with *function variables*. This yields a *schematic* proof which is instantiated subsequently for obtaining proofs of new, similar conjectures. We present techniques for analyzing, generalizing and managing proofs, yielding a proof dictionary of reusable schematic proofs [1, 3]. We also develop techniques for retrieving, solving and patching schematic proofs from a proof dictionary for proving new conjectures by reuse, and show how our reuse proposal supports the discovery of useful lemmata [2, 4].

Bibliography

[1] Thomas Kolbe and Christoph Walther. Reusing Proofs. In A. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence, Amsterdam*, pages 80–84. John Wiley & Sons, Ltd., 1994.

[2] Thomas Kolbe and Christoph Walther. Patching Proofs for Reuse. In Nada Lavrac and Stefan Wrobel, editors, *Proc. European Conf. on Machine Learning (ECML-95)*, pages 303 – 306, 1995.

[3] Thomas Kolbe and Christoph Walther. Proof Management and Retrieval. In *Proceedings of the IJCAI'95 Workshop on* Formal Approaches to the Reuse of Plans, Proofs, and Programs, 1995.

[4] Thomas Kolbe and Christoph Walther. Second-Order Matching modulo Evaluation — A Technique for Reusing Proofs. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence,* Montreal, Canada, 1995.

Certifying Inductive Proofs in NEVER

Sentot Kromodimoeljo
ORA Canada

Bill Pase
ORA Canada

NEVER is the theorem proving component of the verification system EVES. (The language of EVES is called Verdi.) It is an interactive theorem prover with automatic capabilities for inductive, equality and arithmetic reasoning. Since the theorem prover is a complex program with over 10,000 lines of Lisp code, one is justified in questioning the validity of proofs performed.

We address this issue by making NEVER log a proof in terms of simple inferences. A proof log is a sequence of log entries; each entry consisting of an inference rule name, an index to into the subexpression on which the inference rule is to be applied, and parameters for the inference rule. A proof checker can then check the proof by applying the sequence of inferences to the conjecture. (An inference is an equivalence-preserving transformation to the conjecture.) If the result of the sequence of transformations is the expression `(TRUE)`, then the proof is considered to be checked.

Since one of the goals of our proof logging effort was to make proofs be logged in terms of very simple inferences, we did not want induction in NEVER, which is essentially Boyer-Moore induction, to be logged in terms of a complicated induction inference rule. Instead, induction is logged in terms of the application of the strong induction principle, use of termination lemma for a recursive function, and propositional and quantifier reasoning.

Bibliography

[1] R.S. Boyer, J S. Moore, *A Computational Logic*, Academic Press, NY, 1979.

[2] D. Craigen, Reference Manual for the Language Verdi, TR-91-5429-09a, ORA Canada, September 1991.

[3] S. Kromodimoeljo, B. Pase, *Proof Logs*, TR-95-5471-03, ORA Canada, June 1995.

[4] M. Saaltink, A Formal Description of Verdi, TR-95-5482-02, ORA Canada, March 1995

Approaches to Hardware Verification by Term-Rewriting

Wolfgang Küchlin
U. Tübingen

Reinhard Bündgen
U. Tübingen

We give an overview of how term-rewriting techniques can be used for the specification and verification of hardware.

First, following [1], we consider a form of term-rewriting based induction called *inductive completion*. We show how inductive assertions about circuit correctness may be formulated in our language and how they would be proved. We also show the usefulness of algorithms for testing *ground reducibility* in order to check the completeness of circuit specifications.

Second, following [2], we report on experiences with the actual verification of the Sparrow microprocessor which is used in a hardware design course in Tübingen. The circuit design in BLIF (Berkeley Logic Interchange Format), as produced by a standard hardware design tool, was mechanically translated into a term-rewriting system to assure that the axioms closely correspond to the actual hardware.

For each instruction of the processor, the specification leads to equations asserting its correct execution. Further assertions concern correct instruction and argument fetches and the problem that each instruction ends in an instruction fetch state. Most resulting equations are unconditional; the remaining ones are skolemized and transformed into unconditional problems. Our ReDuX prover is then able to prove all theorems using term-rewriting modulo AC. The verification exhibited a few marginal problems with the given implementation of the Sparrow design.

Bibliography

[1] Reinhard Bündgen and Wolfgang Küchlin. Term rewriting as a tool for hardware and software design. In J. Rozenblit and K. Buchenrieder, editors, *Codesign: Computer-Aided Software / Hardware Engineering*, chapter 1, pages 19–40. IEEE Press, 1995.

[2] Reinhard Bündgen, Werner Lauterbach, and Wolfgang Küchlin. A license to fly: Verifying the Sparrow-0 processor. Technical Report WSI-95-4, Informatik, Universität Tübingen, D-72076 Tübingen, 1995.

Inference Rules for Inductive Theorem Proving in Hierarchical Specifications

Ulrich Kühler
Universität Kaiserslautern

Claus-Peter Wirth
Universität Kaiserslautern

Given a hierarchical conditional specification $spec = (sig, R)$ such that the rewrite relation induced by R is ground confluent, we can associate with *spec* its rewrite semantics $\mathcal{I}$, namely the quotient of the ground term algebra and the congruence relation induced by R. $\mathcal{I}$ implies a natural notion of inductive validity: A clause C is inductively valid with respect to *spec* iff every ground instance of C contains at least one literal valid in $\mathcal{I}$.

We present a system of relatively powerful inference rules which can be used to prove inductive validity of clauses as defined in our specification framework. The inference system adheres to the problem reduction paradigm: Each inference rule reduces one goal to zero or more subgoals whose proof is sufficient for that of the goal. In principal, each goal can be used as a possibly open (i. e. unproven) lemma or as an induction hypothesis. Thus, mutual induction is possible. To enable the formulation of the "inductive constraints" (that arise when induction hypotheses are applied to (sub-) goals) we require that each goal is equipped with a weight that measures the "size" of the goal. Weights are compared with respect to a well-founded and stable induction ordering.

In order to keep track of the dependencies among (sub-) goals that result from applications of (open) lemmas and inductive hypotheses, we propose the use of proof trees (or rather proof state trees) and a dependency graph. Our criterion for inductive validity of goals is expressed in terms of these data structures. Moreover, proof state trees form an adequate basis for user interaction and proof analysis.

Induction of Regular Types

David McAllester
MIT AI Laboratory, Cambridge Mass.

There has been considerable recent interest in inferring regular types for functional programming languages and, as a result, considerable interest in set constraints as a modeling tool in static program analysis. The inference of regular types provides theorems of the form $\forall x_1, \ldots, x_n \;\; P_1(x_1) \wedge \ldots \wedge P_n(x_n) \rightarrow Q(f(x1, \;\ldots,\; x_n))$ where P_i and Q are regular predicates. We give a new algorithm for the inference of regular types which is polynomial time in the case where one bounds the number of free variables in program expressions. This polynomial time algorithm allows a form of recursion polymorphism and is more accurate than the Aiken-Wimmers or Heinze systems on broad class of examples.

Automated Verification of Generic Algorithms in the C++ Standard Template Library Using Subgoal Induction

Changqing Wang and David R. Musser
Rensselaer Polytechnic Institute

We present a new approach to the formal verification of programs, called *dynamic verification*, and its application to C++ template-based generic algorithms. The method employs Hoare-style pre/post-condition specifications, symbolic execution based on forward assignment axioms (rather than the usual backward substitution) [MuWa95], and a while-loop inference rule based on subgoal induction [MoWe77]. The symbolic execution mechanism includes multiple Run-Time Analysis Oracles, each of which consists of a C++ interface and a rule-based inference engine.

We briefly describe the MELAS system, which supports the dynamic verification method. MELAS extends a normal debugging system with additional commands for formal verification, normal and symbolic testing, and rapid-prototyping using executable specifications. Since MELAS is an extension of debugging tools many programmers are already familiar with, and it can be applied selectively to small program segments, it should assist in achieving more widespread use of symbolic execution and formal verification technology.

MELAS is still under development, but a preliminary version has sufficient capabilities to formally verify simple generic algorithms taken directly from the ANSI/ISO C++ Standard Template Library.

Bibliography

[MoWe77] J.H. Morris and B. Wegbreit, "Program Verification by Subgoal Induction," in Current Trends in Programming Methodology, R. T. Yeh, ed., Vol. II, Ch. 8, Prentice-Hall, 1977.

[MuWa95] D. R. Musser and C.Q. Wang, A Basis for Formal Specification and Verification of Generic Algorithms in the C++ Standard Template Library, Dept of Computer Science, Rensselaer Polytechnic Institute, Technical Report 95-1, Jan. 1995 (submitted to TOPLAS).

Application of Mathematical Induction to the Formal Proof of Hardware and Embedding in a CAD Framework

Laurence Pierre
LIM-Université de Provence, Marseille

This talk focuses on the use of inductive theorem proving for verifying hardware devices. We emphasize the fact that formal proof tools should be embedded in a CAD system and proposed as an alternative to simulation or test. We give an overview of a prototype proof environment, called PREVAIL, that we are developing in close cooperation with the team of Dominique Borrione in Grenoble [1]. This system takes as input descriptions written in a synchronous subset of VHDL, and verifies the equivalence of (or at least an implication between) two VHDL "architectures" of the same "entity", or checks the validity of temporal properties of an "architecture". It includes a set of special-purpose or general-purpose proof tools, and among them the Boyer-Moore theorem prover, Nqthm.

More precisely, the task of our team in this project consists in evaluating various kinds of general-purpose theorem provers or proof assistants, and in proposing specific modelling and proof methodologies. We are currently focusing on three systems : Nqthm, LP (the Larch Prover), and COQ which is based on the Calculus of Constructions.

The main advantages of Nqthm with respect to formal verification of hardware are its high level of mechanization, its induction and generalization mechanisms, and the possibility of using different kinds of inductive data types with conversion functions (for instance, bit-vectors and natural numbers can be used together in the same proof, and this feature allows one to compare a high-level arithmetic specification with an implementation described at the bit-vector level). Consequently, such an approach can give better results than BDD-based techniques for certain categories of problems. Our former and current works in this field include :

- the definition of recursive functional models for synchronous sequential circuits. We have proposed and mechanically proved equivalent two different models, and we have compared them on simple but significant benchmarks [5],
- the definition of recursive functional models for replicated and parallel architectures. Thus, we have modelled and verified several kinds of one-dimensional or two-dimensional structures (adders, multipliers, etc...) [2],
- the development of specialized methods as pre-processing to the proof, in particular for the generalization of partial specifications [4],
- the proof of the equivalence of two VHDL high-level specifications of the same non-trivial benchmark proposed by Thomsom [3].

The integration of our methods in PREVAIL implies in particular :

- the choice of a synchronous VHDL subset and the definition of an associated semantics,
- the definition of a common intermediate form, for a unified translation from VHDL to the different proof tools,
- the development of a VHDL library of pre-proven standard basic modules.

Bibliography

[1] D. Borrione, L. Pierre, and A. Salem. Formal Verification of VHDL Descriptions in the PREVAIL Environment. IEEE Design and Test Magazine, vol. 9, 2, June 1992.

[2] L. Pierre. VHDL Description and Formal Verification of Systolic Multipliers. In *CHDL and their Applications*, D.Agnew and L.Claesen ed., North Holland, 1993.

[3] F. Nicoli and L. Pierre. Formal Verification of Behavioural VHDL Specifications : a Case Study. In *Proceedings of EURO-DAC'94 with EURO-VHDL'94*. IEEE Computer Society Press, September 1994.

[4] L. Pierre. An automatic generalization method for the inductive proof of replicated and parallel architectures. In *Theorem Provers in Circuit Design, LNCS 901*, R.Kumar and T.Kropf ed., pages 72–91, Springer-Verlag, 1995.

[5] L. Pierre. Describing and Verifying Synchronous Circuits with the Boyer-Moore Theorem Prover. In *Proceedings IFIP WG10.5 Conference CHARME'95*. To appear in LNCS, Springer-Verlag, 1995.

Synthesis and Manipulation of Proof Predicates

Martin Protzen
Technische Hochschule Darmstadt
Fachbereich Informatik

In many cases induction proofs fail because multiple occurrences of variables block the application of induction hypotheses or lemmata necessary to transform the induction conclusion are not available. A solution to this problem is to generalize a conjecture $\forall x^*.\ \Psi$ to $\forall x^*, y^*.\ \Psi'$, i.e. terms from Ψ are replaced by new variables y^*.

However, this approach bears the danger of overgeneralization. Therefore we invent *proof predicates*: from the (incomplete) induction proof of $\forall x^*, y^*.\ \Psi'$ we synthesize the definition of a predicate $G(x^*,\ y^*)$ such that for every instance of $G(x^*, y^*)$ which can be evaluated to TRUE the corresponding instance of Ψ' holds.

Since the original conjecture is an instance of the generalized conjecture $\forall x^*.\ \Psi$ holds whenever $\forall x^*, y^*.\ G(x^*, \sigma(y^*))$ holds.

In general, the proposed technique is able to compute a formula Φ for any invalid (or inprovable) formula Ψ such that the implication $\forall \ldots\ \Phi \rightarrow \Psi$ holds. The converse however, i.e. the implication $\forall \ldots\ \Psi \rightarrow \Phi$, is not true in general.

In some cases the proof of the validity of particular instances of the proof predicate becomes blocked for the same reasons the original proof was blocked. A certain class of these proofs can be unblocked by an equivalence preserving transformation of the proof predicate.

Induction on Non-Freely Generated Data Types

Claus Sengler
DFKI, Saarbrücken

Proofs over recursively defined objects are done using the induction principle which is based on a well-founded order relation.

Non-freely generated data types, that are data types with different syntactical representations for a single object, are frequently used in formal specifications. However, in the area of explicit induction there is a lack in the analysis of these data types in order to obtain an appropriate induction scheme.

We present a method for such a recursion analysis which is based on a very general well-founded ordering for each data type (*minimal size order*). This method is an extension of Walthers estimation calculus with argument-bounded functions. Although we have to restrict the applicable data types to those with *monotonic* constructor functions we are able to handle, for instance, finite sets, arrays, etc. and, of course, all freely generated data types.

Language and Induction

Natarajan Shankar
SRI International Computer Science Laboratory

The Sapir-Whorf hypothesis states that language influences thought. We claim that language certainly influences the effectiveness of automated deduction. In particular, several theorems that are proved by means of difficult induction arguments can be seen as instances of general theorems with straightforward proofs. A sufficiently expressive language is needed to capture such general theorems. We show how a fusion theorem (due to Bird) and a continuation-based transformation (due to Wand) can be stated using the higher-order logic of PVS which supports predicate subtypes, dependent types, and parameterized theories. The PVS proof checker exploits the synergistic interaction between language and deduction to support the construction of elegant proofs of these theorems, as well as those of the binomial theorem and an N-process mutual exclusion protocol (where model checking is used in the induction step).

Reasoning about Hardware Circuits in RRL

Deepak Kapur
SUNY at Albany

M. Subramaniam
SUNY at Albany

Correctness proofs of different adder and multiplier circuits mechanized on an automated theorem prover *RRL* (Rewrite Rule Laboratory) are presented. A main objective has been to minimize user guidance and intervention in the form of additional lemmas typically needed to show that computations at the bit representation level indeed realize the intended arithmetic functions. Two main ideas are exploited. Firstly, different circuits with the same functionality are described using a data structure suited for the circuit architecture. Secondly, functional behavior of circuits is abstracted to describe a generic component with certain behavioral constraints that can be used in other circuits for realizing more complex behavior. The first idea is illustrated using adder circuits. A concise correctness proof of carry lookahead adder circuit using powerlists supporting divide and conquer strategy and parallel prefix computation is described. This circuit is shown to realize addition on numbers by showing its equivalence to ripple carry circuit described using linear lists. The judicious choice of appropriate data structures that enable easy reconciliation of representations underlying the computation and the properties turns out to be a major advantage. The second idea is illustrated by developing a common top level proof for a large class of multiplier circuits including the linear array, Wallace tree and the 7-3 multiplier. By abstracting various adder circuits used in multipliers in terms of generic components with behavioral constraints, we illustrate how the specification and implementation aspects can be decoupled. The use of such generic components would lead to a hierarchical top-down development of proofs of complex hardware circuits.

A Calculus for Rippling

David Basin
Max-Planck-Institut für Informatik
Saarbrücken, Germany

Toby Walsh
IRST, Trento &
DIST, University of Genova, Italy

We present a calculus for rippling, a special type of rewriting using annotations. These annotations guide the derivation towards a particular goal. Although it has been suggested that rippling can be implemented directly via first-order term rewriting, we demonstrate that this is not possible. We show how a simple change to subterm replacement and matching gives a calculus for implementing rippling. This calculus also allows us to combine rippling with conventional term rewriting. Such a combination offers the flexibility and uniformity of conventional rewriting with the highly goal-directed nature of rippling. The calculus we present here is implemented and has been integrated into the Edinburgh CLAM proof-planning system.