An Approach to Flexible Forms of Proof Control for a First-Order Inductive Theorem Prover (Extended Abstract)

Ulrich Kühler

FB Informatik, Universität Kaiserslautern, D-67653 Kaiserslautern, Germany
kuehler@informatik.uni-kl.de

Abstract. We propose an approach to the problem of proof control for our new first-order inductive theorem prover QuodLibet that is characterized by a great deal of flexibility w.r.t. the forms of proof control the prover supports. The approach is based on so-called (proof) tactics, i.e. proof control routines written in a special proof control language named QML. QuodLibet provides a set of tactics (in addition to the elementary inference rules), which range from tactics for trivial simplification steps to tactics representing comprehensive inductive proof strategies. Moreover, QuodLibet allows new tactics that are written by the user in QML to be integrated into the system to dynamically extend its functionality.

1 Introduction

Because of several case studies on the practice of inductive theorem proving in the literature (see e.g. [2], [9], [14]), but also because of our own experience of (automated) inductive theorem provers, such as Nqthm, Inka, RRL, Unicom, Spike etc.[1], we have come to adopt a rather pragmatic view of inductive theorem proving: *The successful use of an inductive theorem prover in "real-life" problem domains has not yet been possible without a knowledgeable human user who can interact with the system on various levels.* This view has mainly determined the concept of proof control which we developed for our new first-order inductive theorem prover named QuodLibet, and which is the subject of this paper.

According to our characterization, the *proof control* of an inductive theorem prover lays down how, speaking in general terms, the construction (or search) of proofs by the prover and/or the user is organized. We distinguish two *basic* forms of proof control, namely user-guided or *interactive* proof control on the one hand and programmed or *automated* proof control on the other hand. Various inductive theorem provers, however, have hybrid forms of proof control. For instance, the Boyer-Moore prover Nqthm is an inductive theorem prover with a complex and elaborate proof procedure forming its essentially automated proof control (see [2]), but there are a few parameters allowing the user to "globally" influence the way the prover searches for proofs — to a limited extent though.[2]

[1] see [3], [10], [11], [8] and [1], respectively
[2] These parameters are the so-called "hints" (see [3]).

Another example of an inductive theorem prover with a hybrid form of proof control is given by LP, the LARCH PROVER (see [6]). Although its proof control is essentially interactive, LP does not require the user to determine every single application of the available (elementary) inference rules: LP provides a small and fixed set of user commands for automatically solving *simple* proof tasks, such as e.g. normalizing terms in conjectures with certain "harmless" (i.e. terminating and unconditional) equations and rewrite rules.

In view of the user's *active* role required for successfully employing an inductive theorem prover, we propose a concept of proof control for QUODLIBET that is to offer a great(er) deal of *flexibility* as to the forms of proof control it supports. Without going too much into detail, let us state more precisely what we require in particular: Our prover is to provide an extensive and easily *extensible* set of user commands that allow the user

- to direct the prover to carry out any possible application of an (elementary) inference rule to any goal in the current proof attempt, thereby facilitating completely interactive proof control on a *low level*
- to invoke comprehensive inductive proof *strategies* in order to obtain complete proofs for simple conjectures constructed by the prover, which realizes (partially) automated proof control on a *high level* and
- to also delegate various (isolated) proof tasks on *intermediate levels* to the prover such as generating induction schemes, doing case analyses based on the definitions of recursive operations, simplifying goals with axioms or lemmas etc.

Observe that we are also interested in enabling the user (i) to modify the intermediate or higher level "inference operators" supplied by the prover or (ii) to add new ones. Thus, we should avoid the disadvantages of "hard-wired" (automated) proof control and be able to utilize QUODLIBET as an *environment for the development of new heuristics and strategies* for the automation of inductive theorem proving. Our overall objective is that QUODLIBET can effectively assist the user in what we consider the *proof engineering* process required for proofs of "real-life" inductive theorems.

2 Logical Foundation of QUODLIBET

Before we can introduce our approach to flexible forms of proof control for QUODLIBET in the next section, we first have to sketch the comprehensive (rewrite-based) *formal framework for inductive theorem proving* which serves as the logical foundation of QUODLIBET. The main components of this framework are (i) an algebraic *specification language* for the formalization of data types, (ii) an *induction order* for guaranteeing applicability of induction hypotheses, (iii) a *calculus for inductive proofs* for formal reasoning about data types and (iv) a concept of a so-called *proof state graph* for the adequate representation of proof constructions. For a systematic and complete presentation of this formal framework we refer to Part I of [12].[3]

[3] The specification language was introduced in [13].

The basis of our formal framework is given by an equational *specification language* which, speaking in general terms, (1) facilitates adequate formalizations of data types with *partial* operations including those that require incomplete or even non-terminating specifications, (2) admits conditional equations (or rewrite rules) with positive and *negative* conditions as axioms and (3) has precisely defined admissibility conditions, which can be easily verified for most practically relevant specifications. In order to briefly explain the other components of the framework for inductive theorem proving, we make use of a simple example (see Figure 1) that is to convey an intuitive impression of how one can construct inductive proofs within the framework.

The inference rules of our *calculus for inductive proofs* reduce goals to (possibly empty) sets of (sub-) goals. Therefore, the notion of a goal is fundamental in our formal framework for inductive theorem proving. A *goal* $\langle \Gamma \,;\, w \rangle$ consists of a clause Γ and a so-called weight w. The purpose of the *weight* w is to provide a measure of the "size" of the clause Γ. By associating weights with clauses in goals we obtain a more flexible and powerful (semantic) *induction order*, which is needed for guaranteeing that every induction hypothesis is of a "smaller size" than the clause it is applied to. So given a conjecture Γ, the tuple comprising the so-called *induction variables* of Γ often yields a suitable weight for Γ. Accordingly, in the case of our (extremely simple) example conjecture, the goal to be proved is $\langle\, \mathsf{plus}(\mathsf{0}, y) = y \;;\; y \rangle$.

Proving inductive theorems in our framework basically amounts to constructing proof state graphs. As can be seen in Figure 1, a *proof state graph* is a directed labeled graph. Each node in a proof state graph is either (i) an *axiom node* labeled with a goal $\langle \Pi \,;\, () \rangle$ for an axiom Π, (ii) a *goal node* labeled with any goal or (iii) an *inference node*, which is labeled with information describing a particular inference step (in Figure 1 the name of the applied inference rule). Every proof state graph initially consists of axioms nodes for the axioms of the specification and of single goal nodes for the conjectures to be proved, and it grows by applications of inference rules to goal nodes. The purpose of a proof state graph is to record the essential dependencies among the (sub-) goals in a proof construction. Such a proof dependency can arise from the reduction of a goal to subgoals with an inference rule or from the application of a goal to another goal as an induction hypothesis or as a lemma. In the latter case, the arc connecting the inference node with the node of the applied goal is labeled with $\mathcal{I}$ or $\mathcal{L}$, respectively.

For instance, in the construction of the proof state graph shown in Figure 1 there was an application of the inference rule Inductive Rewriting to the goal $\langle\, \mathsf{s}(\mathsf{plus}(\mathsf{0}, z)) = \mathsf{s}(z) \;;\; \mathsf{s}(z) \rangle$ that yielded the two subgoals $\langle\, \mathsf{s}(z) = \mathsf{s}(z) \;;\; \mathsf{s}(z) \rangle$ and $\langle\, z < \mathsf{s}(z) \vee z = \mathsf{plus}(\mathsf{0}, z) \vee \mathsf{s}(\mathsf{plus}(\mathsf{0}, z)) = \mathsf{s}(z) \;;\; \mathsf{s}(z) \rangle$ (the latter one is a so-called *order subgoal*) so that a new inference node and two new goal nodes were generated. Since this inference step also involved the use of the goal $\langle\, \mathsf{plus}(\mathsf{0}, y) = y \;;\; y \rangle$ as induction hypothesis, the proof state graph contains an $\mathcal{I}$-labeled arc leading from the new inference node to the goal node for the conjecture.

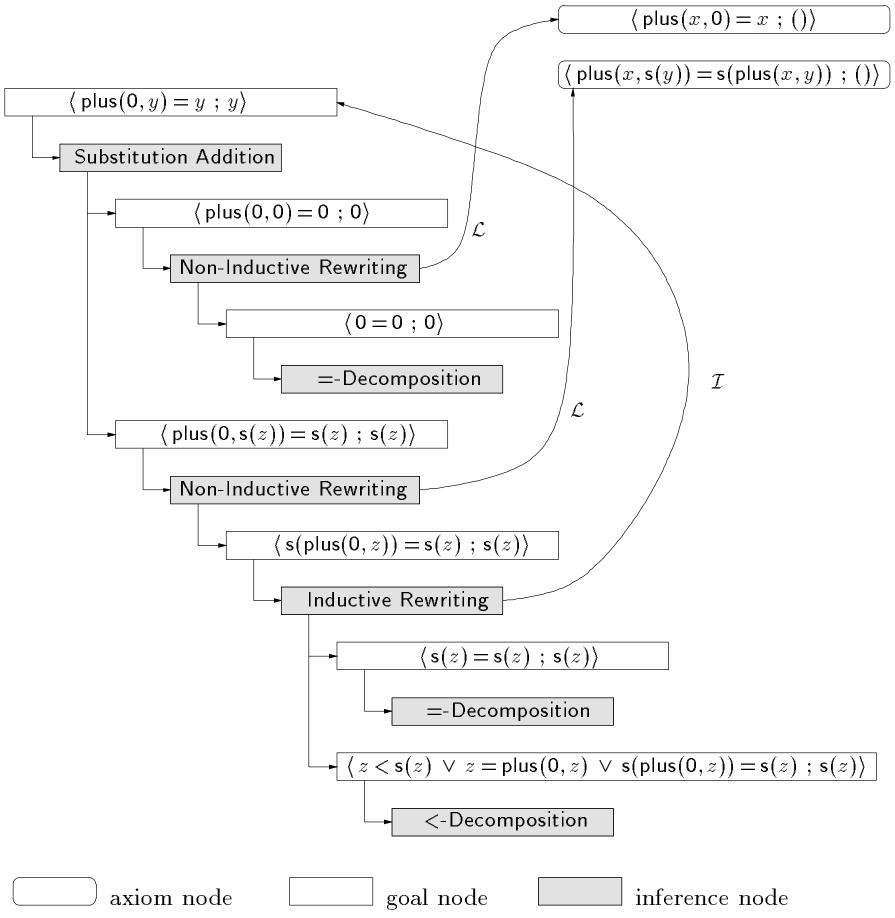

Fig. 1. A proof (state) graph for the inductive theorem $\mathsf{plus}(0, y) = y$

So in order to obtain a proof of an inductive theorem Γ in our framework, a proof state graph needs to be constructed that contains a so-called *proof graph* for a goal of the form $\langle \Gamma\, ;\, w\rangle$. Essentially, a proof graph for a goal $\langle \Gamma\, ;\, w\rangle$ can be thought of as a subgraph P of the constructed proof state graph which represents a complete or "closed" proof attempt for $\langle \Gamma\, ;\, w\rangle$ — closed in the sense that there are no "open" goal nodes in P and that all the lemmas "used in P" are proved. Evidently, the proof state graph in Figure 1 is also a proof graph for the goal $\langle\, \mathsf{plus}(0, y) = y\ ;\ y\rangle$.

In [12] it is shown that by incorporating proof state graphs, our framework for inductive theorem proving is capable of supporting (1) the delayed or *lazy* generation of induction hypotheses, (2) *mutual induction* for conjectures about mutually recursive operators, (3) applications of *unproved* lemmas as well as (4) *multiple* complete or incomplete proof attempts for the same formula.

3 Tactic-Based Proof Control in QuodLibet

The central idea underlying QuodLibet's approach to proof control is that flexibility with regard to proof control as required in the introduction should be achievable on the basis of so-called (proof) *tactics*. Observe that there have been other (mostly higher-order) theorem proving environments with tactic-based approaches to (partially) automating proof constructions, such as e.g. LCF, Nuprl or Oyster-Clam (see [7], [5] and [4], respectively). Our concept of a tactic, however, essentially differs from each of the ones belonging to the just mentioned systems, as will become apparent in the following (to those familiar with these systems).

A first essential characteristic of the proposed tactic-based approach to proof control is that it lays down fundamental properties of the *software architecture*[4] of our inductive theorem prover, and thereby affects the later (system) design phase as part of the process of developing QuodLibet. To be more precise, our approach determines (1) the top level decomposition of the system into the following three main components or subsystems, namely (i) a subsystem for the two types of user interfaces, (ii) a *proof control unit* as well as (iii) an *inference machine* (see Figure 2); and (2) the data and control flow between these subsystems as presented in Figure 2. In addition to this, our approach makes some general requirements with regard to the inference machine:

- All the data objects representing the current specification and the current proof state graph are to be stored in (data units of) the inference machine.
- In order to effect any activities of the system which actually bring about changes of the current specification or of the current proof state graph, both the proof control unit (see below) and the user interfaces subsystem are compelled to generate calls of suitable *inference machine operations*. These operations are offered as resources at the interface of the inference machine and include operations for (i) introducing sorts, operators and axioms, (ii) creating so-called proof state trees consisting of single goal nodes for conjectures and (iii) *single* applications of (elementary) inference rules.
- In particular, the choice of inference machine operations has to ensure that *every* expansion of a goal node in the current proof state graph be the result of an application of an inference rule.

An important consequence of these architectural requirements is that our approach to proof control does not necessitate the "verification" of tactics. That is, every single change of the current proof state graph which is brought about by the proof control unit during the execution of any tactic is *legal* in the sense that the inference machine is required to admit only steps during the construction of inductive proofs that comply with the definition of a proof state graph.

[4] Roughly stated, the *software architecture* of a software system describes the complete (recursive) decomposition of the system into subsystems down to the level of modules, and it defines the interfaces of the subsystems and modules making up the software system.

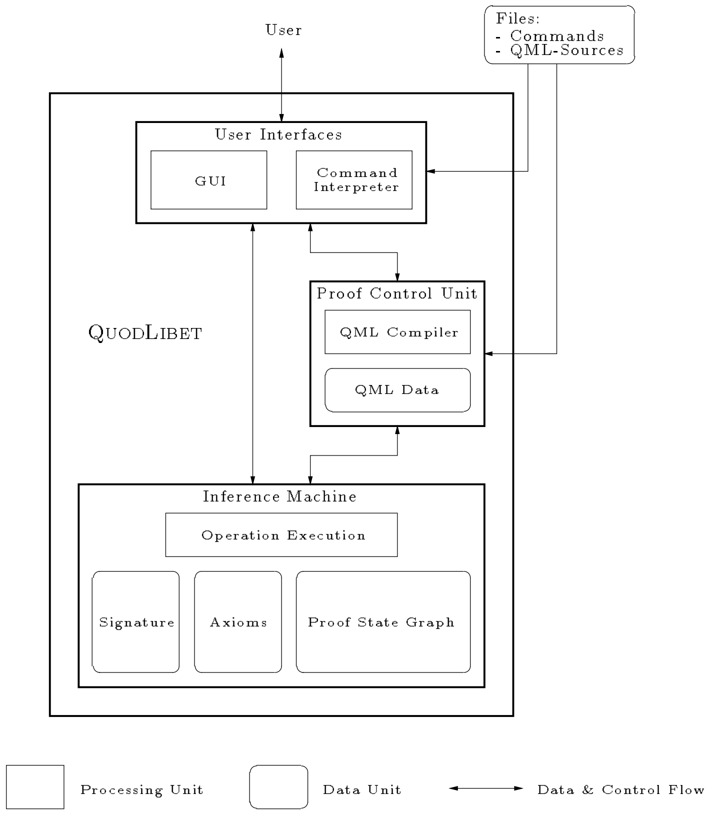

Fig. 2. High level system structure of QuodLibet

In other words, the inference machine can be considered the "secure" kernel of the prover.

A second major component of our tactic-based approach to proof control is the so-called *proof control unit* (PCU). This subsystem of QuodLibet constitutes the setting for the realization of the flexible forms of interactive and automated proof control. Against the background of its high level system structure (see the discussion above), it should be clear that in the case of QuodLibet, (automated) proof control virtually means (automated) control of the inference machine during the construction of inductive proofs. Therefore, the basic purpose of the PCU may be seen as controlling the inference machine over *several*, in some cases even over a large number of proof steps. As will become evident in the following, the control of the inference machine by the PCU is *programmed*. That is to say, when a (higher level) inference operator provided by the PCU, such as e.g. an inductive proof strategy, is being applied to a proof problem, the PCU executes a corresponding *proof control routine* or tactic, thereby generating calls of inference machine operations as specified in the proof control routine. For reasons to be explained below, we have developed a special proof control language named QML for implementing such proof control routines.

In order to characterize the user's view of the proof control unit, let us briefly deal with the system functions which the PCU itself[5] makes available to the user.

- The PCU includes a compiler allowing the user to translate definitions of (new) QML routines, i.e. proof control routines written in QML, into code executable by the PCU.
- Moreover, the PCU offers the user system functions for loading/removing the code of previously compiled QML routines into/from the data unit of this subsystem as shown in Figure 2.
- Last but not least, the PCU can also be employed to execute the code of QML routines, as was mentioned above.

By making use of this functionality, the user may, at any time, alter the set of QML routines that are accessible through the PCU, and thus we do not regard these proof control routines as part of the PCU. Hence, the PCU (along with QML) represents merely the *setting* for realizing the automation of the proof control of QuodLibet.

A third main feature of our tactic-based approach to proof control is given by the proof control language *QML* (*Q*uodLibet *M*eta*L*anguage). Experiences by other authors, such as e.g. Boyer and Moore (see [2] and [3]), establish that developing an inductive theorem prover with sufficiently efficient automated proof control amounts to a complex undertaking that includes a considerable *programming* effort. Because of this, we decided to use the widely known and well-established programming language Pascal as the starting point for the development of QML. Thus, QML essentially resembles an imperative higher level programming language, which, however, is especially adapted to the needs of programming the control of QuodLibet's inference machine. One of the major requirements to guide the development of QML was to create a language that is particularly suited for the (comparatively) easy and convenient implementation of proof control routines by as many users as possible. The subsequent properties of QML contribute to our attempt at meeting this requirement:[6]

- There are predefined data types for the objects of the *object language* of QuodLibet, namely terms, weights, literals, clauses, goals, proof state trees etc. Moreover, QML routines may call upon certain inference machine operations, e.g. upon those for applying inference rules.
- QML enables the user to conveniently define and make use of new (recursive) data types without using pointer types.

[5] This does not include the proof control routines that are provided as part of QuodLibet (see below), although strictly speaking, they also add to the functionality of the PCU.

[6] Due to lack of space, we have to refer to [12] for a complete description of QML, which also includes the QML code for a few example tactics.

– Besides *tactics*, i.e. QML routines intended to expand goal nodes by applications of inference rules, QML also admits "proof control" routines that never actually effect a change of the current proof state graph. An example of such a QML routine could be one for analyzing (recursive) axiomatizations of operators.
– QML features a series of control structures including a conditional construct and various iteration facilities. Furthermore, (recursive) calls of QML routines may occur in QML routines.
– QML provides a useful mechanism for handling so-called *exceptions*, which e.g. may be due to the non-applicability of an inference rule during the execution of a tactic.
– Last but not least, QML requires a modular organization of the proof control routines to be made available to the user through the PCU. The QML compiler expects the source code of exactly one QML module in every input file.

A crucial consequence of these properties is that by means of our proof control language, we achieve an easily comprehensible *abstraction* from the details of the design and implementation of (the inference machine of) QuodLibet. Thus, QML allows not only the system developers, but virtually *any* user who has done some programming in an imperative language, to participate in developing the proof control of QuodLibet by modifying available QML modules or adding new ones.

It has already been emphasized that the PCU (on the basis of QML) constitutes merely the setting for implementing the desired flexible forms of interactive and automated proof control. Consequently, our approach to proof control would not be comprehensive without the QML modules which we provide as part of QuodLibet. These modules make up the final major component of our approach and represent the "intelligence" of our inductive theorem prover in that they realize the required (partial) automation of the proof control of QuodLibet. In order to convey a rough impression of the additional functionality of the system which results from the provided QML modules, we now sketch the so-called *public* QML routines of these QML modules, i.e. those QML routines that the user may explicitly invoke through the PCU.

– There are QML routines for initializing, extending and viewing the (proof control) *database*. Entries into this database can be brought about (i) by a QML routine that analyzes the definition scheme of a given operator of the current specification and tries to determine its domain, or (ii) by a QML routine that classifies a given proved conjecture with regard to possible uses as a lemma.
– One of our major (QML) tactics may be applied to perform a complete inductive case analysis for the clause of a given goal (node), whereas the call of a related tactic yields a case analysis based on the expansion of just one given operator in the goal. Both tactics are guided by the information on the definition schemes of the involved (recursive) operators as stored in the database.

- We provide tactics for various simplification tasks, such as (i) recognizing tautologies or removing redundant literals, (ii) simplifying clauses with axioms and lemmas by rewriting and subsumption, or (iii) proving goals whose "leading" literals are certain definedness or order atoms.
- Fully automated proof control during the construction of inductive proofs for simpler conjectures is achieved with our *strategy tactic* (a restricted, usually terminating variant is also provided), which integrates the just mentioned tactics into a generally applicable inductive proof procedure.

Right now, we provide seven QML modules along with QuodLibet that realize a total of six public procedures and twelve public tactics (see [12] for details and interesting example applications). All in all, these modules consist of more than 4,500 lines of (QML) code and contain the definitions of approximately 130 (auxiliary) routines.

4 Conclusion

In this paper, we have proposed a novel solution to the problem of proof control for our first-order inductive theorem prover QuodLibet that is characterized by a great deal of *flexibility* with regard to the forms of proof control the prover supports. This approach is based on (proof) tactics, for which we developed a special proof control language named QML. Since QuodLibet provides (in addition to the elementary inference rules) various tactics ranging from "intermediate level" tactics for simplification tasks to "high level" tactics representing comprehensive inductive proof strategies, we achieve a useful compromise between ("mindless") completely interactive proof control on the one hand and ("hard-wired") automated proof control on the other hand. Moreover, practical experience of programming in QML has shown that it is comparatively easy to extend existing (strategy) tactics or to write new ones in order to implement newly developed proof heuristics. For example, it required little effort to derive the restricted, usually terminating strategy tactic (see above) from the general strategy tactic that we developed before.

References

1. Adel Bouhoula and Michaël Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14:189–235, 1995.
2. Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, New York, 1979.
3. Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.
4. Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *Proc. of the* 10^{th} *International Conference on Automated Deduction*, volume 449 of *LNCS*, pages 647–648. Springer, 1990.
5. Robert L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

6. Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, DEC-SRC, 1991.
7. M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
8. B. Gramlich and W. Lindner. A guide to UNICOM, an inductive theorem prover based on rewriting and completion techniques. SEKI-Report SR-91-17, Fachbereich Informatik, Universität Kaiserslautern, Germany, 1991.
9. Bernhard Gramlich. Completion based inductive theorem proving: A case study in verifying sorting algorithms. SEKI-Report SR-90-04, Fachbereich Informatik, Universität Kaiserslautern, Germany, 1990.
10. Dieter Hutter and Claus Sengler. INKA: The next generation. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the* 13^{th} *International Conference on Automated Deduction*, volume 1104 of *LNAI*, pages 288–292. Springer, 1996.
11. Deepak Kapur and M. Subramaniam. New uses of linear arithmetic in automated theorem proving by induction. *Journal of Automated Reasoning*, 16:39–78, 1996.
12. Ulrich Kühler. A tactic-based inductive theorem prover for data types with partial operations. PhD Thesis, Fachbereich Informatik, Universität Kaiserslautern, Germany, 1999. Forthcoming.
13. Ulrich Kühler and Claus-Peter Wirth. Conditional equational specifications of data types with partial operations for inductive theorem proving. In H. Comon, editor, *Proc. of the* 8^{th} *International Conference on Rewriting Techniques and Applications*, volume 1232 of *LNCS*, pages 38–52. Springer, 1997.
14. Hantao Zhang and Xin Hua. Proving the chinese remainder theorem by the cover set induction. In D. Kapur, editor, *Proc. of the* 11^{th} *International Conference on Automated Deduction*, volume 607 of *LNAI*, pages 431–445. Springer, 1992.