How to Prove Inductive Theorems? QuodLibet!

Jürgen Avenhaus[1], Ulrich Kühler[2], Tobias Schmidt-Samoa[1], Claus-Peter Wirth[3]

[1] FB Informatik, Univ. Kaiserslautern {avenhaus,schmidt}@informatik.uni-kl.de
[2] sd&m AG, D-40880 Ratingen Ulrich.Kuehler@t-systems.com
[3] FR Informatik, Saarland Univ., 66123 Saarbrücken wirth@logic.at

1 Why Another Inductive Theorem Prover?

QuodLibet is a tactic-based inductive theorem proving system that meets today's standard requirements for theorem provers such as a command interpreter, a sophisticated graphical user interface, and a carefully programmed inference machine kernel that guarantees soundness. In essence, it is the synergetic combination of the features presented in the following sections that makes QuodLibet a system quite useful in practice; and we hope that it is actually *as you like it*, which is the Latin "quod libet" translated into English. We start by presenting some of the design goals that have guided the development of QuodLibet. Note that the system is not intended to pursue the push bottom technology for inductive theorem proving, but to manage more complicated proofs by an effective interplay between interaction and automation.

1.1 Design Goals for Specifications

Given algebraic specifications of algorithms in the style of abstract data types, we want to prove theorems even if the specification is not (yet) sufficiently complete. As an example, consider the incomplete specification of the subtraction on the natural numbers $E = \{\forall x.\ x - \mathsf{0} = x,\ \ \forall x, y.\ \mathsf{s}(x) - \mathsf{s}(y) = x - y\}$ and the conjecture $\forall x, y.\ (x - y = \mathsf{0} \wedge y - x = \mathsf{0} \Rightarrow x = y)$. Note that this is indeed an inductive theorem with respect to all consistent extensions of the given specification.

For a more complicated example, assume we want to prove that the lexicographic path order *lpo* is totally defined on ground terms and is indeed an order. QuodLibet admits the natural definition of *lpo* by mutual recursion.

All in all, we propose a specification formalism for our inductive theorem prover that admits *partial* definitions of operators over free constructors using (possibly *non-terminating*) *positive/negative*-conditional equations as well as *destructor* recursion or *mutual* recursion. A conjecture may be any (universally quantified) clause. If it is inductively valid with respect to a specification it should also be valid in all consistent extensions of the given specification, i.e. in our approach inductive validity is monotonic with respect to consistent extensions.

1.2 Design Goals for Proving Theorems

QuodLibet's inference system is intended to formalize techniques commonly used by mathematicians, especially when applying induction hypotheses. In *explicit* induction (cf. [7]) as found e.g. in Nqthm [2], an *induction rule* is provided whose addition turns a deductive into an inductive inference system without further changes on the deductive part. Roughly speaking, explicit induction "hides" several (applications of) induction hypotheses in a single inference step. Contrary to this, however, many mathematicians perform inductive proofs as follows:

They begin with the conjecture and simplify it by case analysis. Realizing that subgoals have become similar to different instances of the conjecture, they apply the conjecture just like a lemma, but keep in mind that they actually applied some induction hypothesis. Finally, they search for a wellfounded order with respect to which all the instances of the conjecture applied as induction hypotheses are smaller than the original conjecture itself.

In this way (besides simulating explicit induction) we intend to construct inductive proofs with QuodLibet; for more details cf. [4]. In [8] this aspect of implicit induction is called *descente infinie*, a name already coined by Fermat. Another motivation for *descente infinie* is to overcome the limitations of recursion analysis as described in [6], where *descente infinie* is called the *lazy method*.

Besides, mathematicians often make different proof attempts, switching from one attempt to another if they get stuck, until they succeed. They introduce lemmas which they only prove if they turn out to be useful. QuodLibet is to support this (tentative) style of *proof engineering*. However, it also needs to be capable of proving simpler theorems and subgoals without any user interaction.

2 QuodLibet Specifications

Given a QuodLibet specification $spec = (sig, E)$ that comprises a signature $sig = (S, C, F)$ where S is the set of sorts and $C \subseteq F$ the set of free constructors, we need to fix the intended semantics of *spec*. We begin by defining E-equality. This is not trivial since E may contain equations (or rewrite rules) with positive and *negative* conditions of the form $t_1 \neq t_2$. A term t is called *defined* if it is E-equal to a constructor ground term $t^C \in \mathcal{T}(C)$. To define E-equality as for Horn-clause specifications we evaluate the negative conditions $t_1 \neq t_2$ constructively: This condition is satisfied if t_1 and t_2 are defined and t_1^C, t_2^C are not equal. We now define $=_E$ as usual and consider the quotient algebra given by $\mathcal{T}(F)/{=_E}$. In our approach, partiality of a function $f \in F$ relates to the *data* of *spec* as given by $\mathcal{T}(C)$. In this view the quotient algebra $\mathcal{T}(F)/{=_E}$ contains a partial algebra $\mathcal{M} = \mathcal{M}(spec)$ with $\mathcal{T}(C)$ as its universe and partially defined functions $f^{\mathcal{M}}$ for the $f \in F$. We call $\mathcal{M}(spec)$ the *standard model* of *spec*. Technically, for each undefined term, $\mathcal{T}(F)/{=_E}$ contains an error element. Two undefined terms t and t' represent the same error element only if $t =_E t'$.

A first attempt to define the semantics of *spec* is validity in $\mathcal{M}(spec)$. However, this semantics is not monotonic with respect to consistent extensions. Hence we propose another semantics (cf. [5, 9]): A first-order model of *spec* (with partial functions) is a *data model* of *spec* if its C-reduct is isomorphic to the free term algebra $\mathcal{T}(C)$. A clause is an *inductive theorem* of *spec* if it is valid in all data models of *spec*. Given this semantics, inductive validity is monotonic with respect to consistent extensions. For example, the claim

$$\forall x, y.\ (x \geq y = \mathsf{true} \wedge y \geq x = \mathsf{true} \quad \Rightarrow \quad x = y)$$

in the specification $E = \{\forall x.\ x \geq 0 = \mathsf{true},\ \ \forall x, y.\ \mathsf{s}(x) \geq \mathsf{s}(y) = x \geq y\}$ looks very similar to the one in § 1.1, but—contrariwise—is not an inductive theorem as $\forall y.\ 0 \geq y = \mathsf{true}$ (which makes $\geq$ trivial) is a consistent extension of E.

QuodLibet provides easily testable admissibility conditions which guarantee that the semantics outlined above is well-defined. These conditions do not require termination of E; instead they essentially require that E fulfills a simple syntactic confluence criterion (cf. [4, 5]).

Finally, QuodLibet supports two kinds of mutual dependencies. Firstly, the data types represented by the sorts S do not have to be hierarchically ordered so that we can define e.g. terms and term lists which mutually depend on each other. Secondly, operators can be defined by mutual recursion. Of course, mutual recursion can be encoded by non-mutual recursion. But that leads to unnatural definitions and, more importantly, to technically more complex proofs.

3 Proving Theorems with QuodLibet

Descente Infinie To realize *descente infinie*, QuodLibet supplies the user with inference rules for inductive case analysis, inductive rewriting and inductive subsumption. Thus, clauses can be used inductively provided they are somehow smaller than the clause they are applied to. In QuodLibet the size of a clause is measured by a so-called *weight* which is an n-tuple of terms associated with the clause. The pair consisting of a clause and a weight is called a *goal*.

To compare goals, QuodLibet uses on the one hand a fixed order on defined terms and on the other hand a flexible scheme to define the induction order: Any admissible specification defines the "semantic length" for any defined ground term t as the length of the unique constructor ground term t^C equal to t. The defined terms are ordered by their semantic length; the lexicographic extension of this order is used to compare weights. The flexibility of the scheme is achieved by the fact that the weights can be composed of any tuple of defined terms.

Using QuodLibet the choice of an appropriate induction order can be delayed until it has to be made. This is done in the following way: Any conjecture is initially equipped with a weight term of the form $w(x_1, \ldots, x_k)$, where w is a new free (existential) variable and $x_1, \ldots, x_k$ are the (universal) variables in the conjecture. Whenever the conjecture is instantiated (e.g. by a cover set of substitutions) the weight is instantiated accordingly. Any inductive application of a clause to the actual conjecture creates a proof obligation of the form $w_1 < w_2$ where w_1 and w_2 are the weights of the applied clause and the conjecture, respectively. During the proof process the function variables w have to be instantiated by functions evaluating to n-tuples of defined terms so that the proof obligations can be fulfilled where $<$ is interpreted as the lexicographic extension of the semantic order given by the specification. As an example, consider the natural specification of mergesort by *destructor recursion*. Here a list l is split into $\mathsf{even}(l)$ and $\mathsf{odd}(l)$, the lists of elements at even and odd positions in l. If one defines $\mathsf{length}(l)$ as the length of l, then one can prove $\mathsf{length}(\mathsf{even}(l)) < \mathsf{length}(l)$ and $\mathsf{length}(\mathsf{odd}(l)) < \mathsf{length}(l)$ for lists l of length greater than 1. This will solve all proof obligations created in the inductive proof for correctness of mergesort.

Note that almost all automated inductive theorem provers derive the induction order from the order used for the termination proof of the specified operators. QuodLibet does not require termination of the specified operators, and the induction order can be constructed more flexibly so that it satisfies the order constraints that arise in the proof. Moreover, no special inference rule is needed to cope with mutual recursion.

Representation of Proof Attempts and Open Lemmas QuodLibet has a sophisticated graphical user interface which enables the user to easily create specifications, manages the already proved lemmas, and substantially supports proof engineering by visualizing proof constructions (cf. [4]).

In the simplest case a proof attempt is represented graphically by a proof (state) tree consisting of inference and goal nodes. An inference node represents the inference applied to its parent which is a goal node. Its n children ($n \geq 0$) are again goal nodes and represent the new goals created by the inference. A goal node is marked with a clause which is to be proved inductively valid, and a weight. Extending the proof attempt means to apply an inference to one leaf which is a goal node. If there is no such open leaf any more then the (proof) tree represents a proof of all the clauses in the goal nodes.

Furthermore, it is also possible to start several proof attempts in parallel as described in [3]. To do so, a goal node may have several inference nodes as children. Any of these subtrees represents a proof attempt. Hence in general one can construct an and/or-tree to represent the proof construction so far. One can work on the different proof attempts independently, just as it seems most promising to achieve a complete proof. So neither replay nor backtracking is needed. All the proof attempts are at the disposal of the user.

As already explained in § 1.2 it is sometimes very useful to conjecture a lemma and to try to get the proof completed by using this lemma. QuodLibet supports this way of constructing proofs. Using the *assume*-command a yet unproved lemma can be introduced for constructing a proof. But this proof is complete only if all assumed lemmas are finally proved. This is controlled by managing the proof dependencies among all used lemmas.

Tactics We have already mentioned that QuodLibet can be used as an interactive theorem prover. However, it would be extremely tedious to determine every proof step manually. Therefore, QuodLibet provides proof *tactics* to automate proof constructions. A tactic is a routine formulated in QML, a new proof control language, which controls QuodLibet's *sound* inference machine kernel. QuodLibet provides a variety of tactics that perform routine proof constructions. We give some examples.

Provided the definition analysis of the functions is already done, a tactic may start the proof of a conjecture by performing an appropriate case analysis, simplify the clauses with defining equations and lemmas and also apply inductive inferences. There is a tactic to guess an appropriate instantiation of the weight variable (i.e. to define an appropriate induction order) and to solve the order-goals. This tactic is in almost all cases successful where the subterm order is enough to prove the order goals. Otherwise, the user is asked to define the induction order by instantiating the weights. The so-called *standard tactic* combines these aspects into one tactic. If it gets stuck it generates a new proof tree and tries to solve the goal by another inductive proof. QuodLibet is an open proof system in the sense that more tactics can be written in QML and easily integrated by the user in order to reduce the need of human interaction.

4 Development History

Looking for a formal inductive calculus for *descente infinie*, the "implicit induction" of [1] (as implemented in Unicom) was a starting point because it included induction hypothesis application; but it was restricted to first-order universally quantified unconditional equations and was not human-oriented. These limitations were so severe in practice that we decided to stop the development of Unicom and use the gained experience for the development of QuodLibet as a system based on a human-oriented inductive calculus for first-order universally quantified clausal logic. So far, the main focus of QuodLibet has been to provide a natural style for specifications and proof techniques. We have used QuodLibet for proofs on natural numbers (e.g. non-terminating specification of division), sorting algorithms on lists (bubblesort, insertionsort, mergesort, quicksort), binary search trees (cf. [4]) and orders (*lpo*). For some more elaborated examples the underlying tactics will have to be improved in combination with the use of decision procedures. A further development along [8] is planned.

In [4] the concepts and the implementation of the whole QuodLibet system are described in detail. The current version of QuodLibet can be obtained from `http://agent.informatik.uni-kl.de/quodlibet.html`.

References

[1] Leo Bachmair (1988). *Proof By Consistency in Equational Theories.* 3rd Annual IEEE Symposium on Logic In Computer Sci. (LICS), pp. 228–233, IEEE Press.

[2] Robert S. Boyer, J Strother Moore (1988). *A Computational Logic Handbook.* Academic Press.

[3] Deepak Kapur, David R. Musser, and X. Nie (1994). *An Overview of the Tecton Proof System.* Theoretical Computer Sci. **133**, pp. 307–339, Elsevier, Amsterdam.

[4] Ulrich Kühler (2000). *A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations.* PhD thesis, Infix, Akademische Verlagsgesellschaft Aka GmbH, Sankt Augustin, Berlin.

[5] Ulrich Kühler, Claus-Peter Wirth (1996). *Conditional Equational Specifications of Data Types with Partial Operations for Inductive Theorem Proving.* 8th RTA 1997, LNCS 1232, pp. 38–52, Springer, Berlin.

[6] Martin Protzen (1994). *Lazy Generation of Induction Hypotheses.* 12th CADE 1994, LNAI 814, pp. 42–56, Springer, Berlin.

[7] Christoph Walther (1994). *Mathematical Induction.* In: Dov Gabbay, C. J. Hogger, J. Alan Robinson (eds.). *Handbook of Logic in Artificial Intelligence and Logic Programming*, Clarendon Press, Oxford, 1993ff., Vol. 2, pp. 127–228.

[8] Claus-Peter Wirth (2004). *Descente Infinie + Deduction.* Logic J. of the IGPL **12**, pp. 1–96, Oxford Univ. Press. `http://www.ags.uni-sb.de/~cp/p/d (Sept. 12, 2003)`.

[9] Claus-Peter Wirth, Bernhard Gramlich (1994). *A Constructor-Based Approach for Positive/Negative-Conditional Equational Specifications.* J. Symbolic Computation **17**, pp. 51–90, Academic Press (Elsevier), San Diego (CA).