Disguising recursively chained rewrite rules as equational theorems, as implemented in the prover EFTTP Mark 2 ⋆

M. Randall Holmes

Boise State University, Boise, Idaho, USA

1 Introduction

This paper describes an approach to writing a tactic language for an equational theorem prover which is implemented in the author's theorem prover EFTTP Mark 2 (research on this prover is supported by a grant from the US Army Research Office, to which the author is very grateful). Since the official name of the prover is long, we abbreviate it as Mark2 hereafter.

Mark2 is written in ML (SML/NJ and Caml Light) and we expected at the outset to follow the lead of other provers (HOL,Nuprl; see [3], [1], respectively) in using ML to write tactics. This turned out not to be needed. Tactics in Mark2 are expressed as equational theorems. They are proven in the same way that any equation is proven and they are stored in the same way as other theorems. An interesting side effect of this is that Mark2 is independent of ML; we are in the process of implementing it in C++ with an eye to improving efficiency, which would not be possible if we were dependent on ML for tactic writing.

The source code for the SML/NJ version of the prover and a limited manual are available at (http://math.idbsu.edu/faculty/holmes.html); the author may be contacted at (holmes@math.idbsu.edu).

2 A Brief Introduction to the Mark2 Prover as a Dumb Equational Prover

The philosophy of the Mark2 prover is to implement purely algebraic reasoning with as few distractions as possible. Purely equational reasoning is as powerful in principle as the apparently more complex forms of reasoning embodied in first order and higher order logics, in the presence of suitable axioms (see, for example, [8], [7], [2] or our [5]). The premise of our project is that, in the presence of automated assistance, purely equational reasoning should prove as effective in practice as other logical frameworks. We do not claim to have proven our point with the present implementation of Mark2 (which lacks various obvious optimizations)!

We designed the prover to keep the notion of substitution as simple as possible. We avoid the use of bound variables. There is no built-in system of absolute

⋆ Supported by the U. S. Army Research Office, grant no. DAAH04-94-G-0247 types, although the definition facility enforces a system of relative typing analogous to that in Quine's "New Foundations" (see our [4] or unpublished [6]).

The syntax of the input language is straightforward; we note that infix precedence is not supported at the moment (all operators associate to the right as far as possible); we will supply parentheses for clarification that the prover itself might not give. Two notational conventions which are not usual are noted: function application is represented by the infix @, and the constant function with value a term T is represented by $[T]$. Variables begin with question marks.

A session with the prover (in its simplest form) begins with the user entering a term, intended to serve as the left side of a theorem to be proved, then proceeding to apply equational theorems as rewrite rules until a final term is reached, at which point the user issues a command recording a theorem, which will then be usable in the same way as the theorems already available.

One problem with this very basic kind of proving is the difficulty of controlling the application of theorems to subterms. We provide the ability to move around the tree representing the term in order to make this easier. The user has the option of applying a theorem in the current theory as a rewrite rule (in either direction) to the currently accessible subterm (only at the top level, not to any of its subterms) or of moving his attention to a different subterm: the basic commands provided are "move to the left subterm", "move to the right subterm", "move up (to the smallest term properly containing the current subterm)" or "move to the top"; more powerful "movement" commands are also available.

This is a very laborious way to prove theorems. Our original intention was to add tactics written in ML which manipulated subterms in more powerful but still safe ways; this turned out to be unnecessary for the most part.

3 The Transformation of a Dumb Equational Prover into a Programming Environment

The underlying idea which led to the development of the tactic-writing method used in Mark2 is simple. We decided to add a device for introducing names of theorems into terms, to signal our eventual intention of applying a given theorem to a given subterm. Suppose "COMM" is the name of the theorem $?x+?y = ?y+?x$, the commutative law of addition. The term $COMM => (?a+?b)+?c$ would have the same denotation as the term $(?a+?b)+?c$, but would convey to the reader the additional information of our intention to apply the theorem COMM to it; the infix => was introduced to construct this kind of term, along with an infix <= which signals the intention to apply a theorem in reverse. A term like $COMM => ?x*?y$ would have the same denotation as $?x*?y$ but would express the odd intention to apply the commutative law of addition to this term. Our original intention went no farther than to allow the introduction of embedded theorem names at various points in a term prior to issuing a command "execute" which would apply all of the theorems thus embedded in the current subterm (where application was possible). The "execute" command simply removed embedded theorem names which did not apply or which it did not recognize. Note that embedded theorems are applied only to the top level term to which they are attached, not to subterms as is more usual (though this effect can be achieved using the tactic-writing method described below; it usually proves more efficient to exercise some control over what subterms are taken to be targets of the rule).

Unexpectedly, it becomes possible to prove "theorems" with interesting behavior. For example, consider the theorem ZERO: $0+?x = ?x$, which one might expect to find in a theory containing the axiom COMM cited above. One can certainly prove a theorem COMMZERO: $?x + 0 = ?x$, using COMM and ZERO together. But the theorem EITHERZERO: $?x = (ZERO => COMMZERO => ?x)$ is a different matter. Certainly it is true—embedded theorem names have no effect on the values of terms. If the command "execute" is defined so as to aggressively carry out all theorem applications it encounters, including those introduced in the course of applying previously applied theorems then the effect of the theorem EITHERZERO will be to apply the identity axiom for addition in either its left or right form, if appropriate (actually, it is possible that it will be applied twice). This is rather surprising behaviour for what appears to be a single equational theorem.

The following is certainly true: $(0+?x) = (ZEROES => ?x)$ So we can prove this theorem and give it the name ZEROES (a declaration of ZEROES as a prospective theorem is required before the proof). The effect of this "theorem" when applied to a term is to eliminate any number of zeroes added on the left and then (equally importantly) stop. The reason that it continues to be applied as long as it sees a zero added on the left is that it introduces an application of itself each time it is successfully applied. This recursion is well-founded because an application of ZEROES eventually encounters a subterm not of this form and fails to be applicable, whereupon it is simply removed by the "execute" process.

The exact behaviour of such theorems depends on the way in which the "execute" command is implemented. The current version proceeds in a "depth-first" manner, applying every embedded theorem that is present initially or is generated by earlier theorem applications, always applying innermost embedded theorems first. A referee pointed out correctly that there is an analogy between the choice of execution order here and PROLOG implementation issues. Confluence holds, so execution order will not affect results of computations which terminate, but termination and efficiency could be affected by the choice of a different order. We are investigating a different execution order more appropriate for a parallel machine.

Since all steps allowed in the execution of theorems interpreted as programs are applications of theorems in the current theory, programs are safe in the sense that they will not lead the prover to prove false theorems. They may fail to prove *any* theorem, by failing to terminate.

4 An example: the construction of a limited abstraction algorithm

The first major application of this technique was the implementation of abstraction and reduction algorithms. These were needed to support the avoidance of bound variables in Mark2, which requires the use of synthetic abstraction terms instead of λ-terms. We describe the way in which an abstraction algorithm for a limited class of terms is implemented.

Before we describe this algorithm, we need to describe two refinements of the tactic-writing language.

We mentioned a problem which arises with the theorem EITHERZERO defined above: if EITHERZERO is applied to a suitable term (e.g., $0+?x+0$), two successive applications of the identity for addition may occur (on different sides) which is a little untidy. The solution is to introduce variants =>> and <<= of the theorem embedding infixes with the property that the indicated theorem is to be applied only if the immediately preceding theorem application fails. This is used to build lists of theorems to be applied as alternatives, suppressing the danger that more than one of them might be applied in sequence. EITHERZERO would now have the form $?x = (COMMZERO =>> ZERO =>?x)$, where we could be certain that there would be only one application of the identity for addition.

The second refinement is the introduction of parameterized theorems. Consider the theorem CONST which defines the expected behavior of constant functions: CONST: $[?x]@?y =?x$. Now consider application of this theorem in reverse: the result of $CONST <=?a$ is $[?a]@???y$ (the system supples the two additional question marks when it is forced to create a variable, in order to avoid unintended collisions of variables). A more sophisticated version of the converse of CONST would be the following: $(REVCONST@?y) : ?x = [?x]@?y$. The parameterized theorem is treated as a "function" applied to its argument. An example: the term $(REVCONST@2) =>?x$ would become $[?x]@2$. The use of parameters is convenient in other contexts where it is desirable to provide information to a tactic which is not contained in the term to which it is applied. Tactics can take both objects of the theory and other theorems or tactics as arguments; they can also take multiple arguments (in different styles: $(TACTIC@arg1)@arg2$ and $TACTIC@(arg1, arg2)$ are both possible forms).

The tactic ABSTRACT is intended to have the following effect: a term $(ABSTRACT@?x)@T$ (where T is a complex term, usually containing occurrences of $?x$) should take the form $U@?x$ when executed, where U is expected to contain no occurrences of $?x$. One should think of U as $(\lambda ?x)(T)$. If T is not a term of a suitable form, the algorithm may fail, in the sense that U will not be free of occurrences of $?x$. It should also be noted that the argument passed to ABSTRACT does not need to be a variable; one term can be expressed as a function of another term of arbitrary form using the ABSTRACT tactic.

Recall that the infix "@" is used to represent function application. The theory in which ABSTRACT is implemented also includes an identity function

Id, an infix "," implementing pairing, a infix ";" implementing function product (with defining axiom PROD: $(?f; ?g)@?x = (?f@?x), (?g@?x))$ and an infix "@@" implementing composition (with defining axiom COMP: $(?f@@?g)@?x = ?f@(?g@?x)$.

The theorem ABSTRACT has the form $(ABSTRACT@?x) : ?y = (ABSCONST@?x) =>> (ABSCOMP@?x) =>> (ABSPROD@?x) =>> (ABSID@?x) =>?y$. This is not very informative; it tells us that one of a sequence of alternative tactics to which the same argument will be passed will be applied to the target term (which may have any form).

The theorem ABSID first applied has the form $(ABSID@?x) : ?x = Id@?x$, where Id is the identity function. This is an obvious base case of the abstraction algorithm.

The theorem ABSPROD next applied has the form $(ABSPROD@?x) : (?y, ?z) = PROD <= ((ABSTRACT@?x) =>?y), ((ABSTRACT@?x) => ?z))$. The intention is that the recursive applications of ABSTRACT will yield something of the form $PROD <= ((?Y@?x), (?Z@?x))$ which the reverse application of PROD will convert to $(?Y; ?Z)@?x$.

The theorem ABSCOMP next applied has the form $(ABSCOMP@?x) : (?f@?y) = COMP <= (?f@((ABSTRACT@?x) =>?y))$; the intention is that a term of the form $COMP <=?f@(?Y@?x)$ result, which the reverse application of COMP converts to $(?f@@?Y)@?x$. Note that it is assumed that the term matching $?f$ will not contain any occurrences of the term matching $?x$ (this is one of the restrictions on the class of terms for which this algorithm works).

Finally the theorem ABSCONST has the form $(ABSCONST@?x) : ?y = [?y]@?x$ (the same as REVCONST above). It will only be applied if the target term is neither the same as the argument nor composite in the sense of being a pair or a function application. Note that if we did not use the alternative forms of the theorem application infixes, this would *always* be applied, which would not give the desired results!

The full implementation of ABSTRACT uses a built-in "theorem" which automatically generates theorems parallel in form to PROD for other infixes (the theorem for addition would read $(?f :+ ?g)@?x = (?f@?x) + (?g@?x)$, for example). The tactic REDUCE which reverses the effect of ABSTRACT is somewhat simpler to implement.

5 Closing Remarks

The effect of the theorems ABSTRACT and REDUCE has been to make it possible to make complex substitution processes invisible to the user, supporting the avoidance of bound variables. The synthetic abstraction terms constructed by ABSTRACT are fairly readable, since they are exactly parallel in structure to the original term, but this is seldom an issue, because ABSTRACT and REDUCE are most often used together in a way which makes it unnecessary for the user of complex theorems built with their help ever to see a function abstraction term. (We are considering the introduction of bound variables to the prover, to make the notation more readable; we are considering how the presence of variable-binding constructions would interact with the tactic-writing strategy discussed in this paper).

Examples of tactics which we have implemented include a full automatic tautology-checker (not very efficient), as well as algebraic expansion and simplification algorithms for the usual algebra and Boolean algebra. A nice example of a tactic taking another tactic as a parameter is a tactic $DUAL$ for a theory of Boolean algebras which generates the dual of its argument (even if its argument is a tactic rather than a simple equation)!

It may be of interest to note that Mark2 allows one to bind a theorem to a function, so that that theorem is applied automatically wherever that function appears applied to an appropriate number of arguments (determined by the form of the theorem bound to it). This works for higher-order functions as well. Thus functional programming can be simulated under the prover.

References

1. R. Constable and others, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, 1986.
2. H. B. Curry and R. Feys, *Combinatory Logic*, Vol. I, North Holland, Amsterdam, 1958.
3. Mike Gordon, "*HOL*, a Proof Generating System for Higher Order Logic", in *VLSI Specification, Verification, and Synthesis*, edited by Birtwistle and Subrahmanyam, Kluwer, 1987.
4. Holmes, M. R. "Systems of combinatory logic related to Quine's 'New Foundations'". *Annals of Pure and Applied Logic*, vol. 53 (1991), pp. 103-133.
5. M. Randall Holmes, "A Functional Formulation of First-Order Logic 'With Infinity' Without Bound Variables", preprint.
6. M. Randall Holmes, "Untyped λ-calculus with relative typing", preprint.
7. W. V. O. Quine, "Algebraic Logic and Predicate Functors", Bobbs-Merrill, 1971 (booklet).
8. Alfred Tarski and Steven Givant, *A Formalization of Set Theory Without Variables*, American Mathematical Society, Providence, 1988.

This article was processed using the LaTeX macro package with LLNCS style