An Experience with Term Rewriting for Program Verification

Sergei D. Mechveliani ⋆

Program Systems Institute, Pereslavl-Zalessky, Russia,
mechvel@botik.ru

Abstract. We have developed a proof assistant based on many-sorted term rewriting, unfailing completion, and inductive reasoning. We are going to interface it to our computer algebra library. Its application also includes automated program and digital device analysis. It also can be used for generating certificates for proofs and programs, with automatic certificate check. The system and the CA library are implemented in the `Haskell` language.

Key words: automatic equational prover, term rewriting, inductive reasoning, proof certificate.

1 Introduction

Having designed a computer algebra library `DoCon` [Me1], we try to extend this system with the ability of automatic reasoning. In this paper we shortly describe the aims and the design principles of our program system for this project.

1.1 The Aim of the Project

The aim is to develop an efficient proof assistant for providing proof certificates for the areas of (1) mathematics, (2) functional programming, (3) digital device analysis.

We need to keep in mind that the general problem of the proof search is algorithmically undecidable. So far, we presume that a human researcher does the main parts of the proof search, the ones which need more ingenuity, and orders the program assistant to fill the "technical" parts with detailed proof. This process is iterated. This approach should lead to the two benefits:
(1) human effort economy in solving problems, (2) proof certificate.

It is known from mathematical logic that each mathematical proof can be unwind to a sequence of elementary steps. Each elementary step is similar to the following: to superpose two formulae (equations) by substituting appropriate

⋆ This work is supported by the Program of Fundamental Research of the Russian Academy of Sciences Presidium ("Razrabotka fundamentalnykh osnov sozdaniya nauchnoj raspredelennoj informatcionno-vychislitelnoj sredy na osnove tekhnologij GRID").

expressions for universally quantified variables, and to derive by this another formula (Robinson's resolution in mechanized reasoning). A proof certificate is a symbolic code consisting of elementary steps of this kind. This is a matter of a program assistant: to obtain the details of certificate and also to check automatically a certificate of a proof or program provided with such by any other system — if this system supports the certificate standard. A certificate guarantees the truth of the statement and that there is no error in the proof.

But to be really usable, such a proof assistant needs

- to "understand" a high-level object language which is close to the human mathematical language,
- to be able to incorporate and use an algorithm and knowledge library for each domain of application,
- to have a powerful proof search strategy, in order to make the automatic proof search part possibly greater (for we think, in the practice of modern assistants, more than 99 % of the effort is by human).

1.2 About our Approach in General

The object language of our system is of the *many-sorted term rewriting* and also of the predicate calculus. An object *program* (subjected to verification) is represented as a set of rewrite rules. We represent a knowledge about a computation domain in the form of equations and apply the technique of many-sorted term rewriting. The predicate calculus statements also convert to equations (in the proof by contradiction) represented as Boolean terms. This enables refutational proofs via completion. The inductive inference is applied for the proofs in the *initial model*, and it cooperates with completion in a natural way.

As to application to programming: inductive proofs for programs correspond to establishing truth in the *initial model* for a set of equations [Hu:Op].

Introductory reading [K:B], [Hu:Op], [Hsi:Ru], and [Lo:Hi] introduce to term rewriting (TRW) and its *completion* method. [Sti] explains unification modulo associativity and commutativity, which leads to AC-completion (implemented in our prover). [Hsi] describes how TRW (with extension to Boolean terms) is applied to refutational proof in predicate calculus. [Bu:Al] and [Bu2] present explanations about language and a program system `Theorema` for proof assistant which looks as the most advanced modern project of this kind. Another two assistant examples: `Coq`, `Isabelle`.

Our prover and the `DoCon` library are implemented in the `Haskell` language. Our prover is called `Dumatel`. This joke Russian word is taken from the novel "Skazka o troike" by the brothers Strugatsky, and it can be translated as "thinker".

`Dumatel` is a library of `Haskell` functions and structures.

About other projects There exist many prover systems. We pay attention to theoretic principles, preferring to implement them in our own system — due to desired interface to our computer algebra library, and due to other reasons. The main particular points of our prover design are the following.

- The intention to express, as possible, all knowledge via equations and TRW.
- Extension of "unfailing" completion to Boolean terms, with a particular treatment of order on monomials.
- Proof by cases, combined in a special way with completion.
- A particular procedure for the search of a useful lemma for inductive proof.
- The resource distribution approach in the proof search.
- Symbolic representation of a proof search state by an explicit tree data.

We use the following **abbreviations and denotations**:
AC — associativity and commutativity, BT — Boolean term (with `&, xor`);
CA — computer algebra, ground term — a term free of variables;
IL, OL — (respectively) implementation language and object language;
TRW — term rewriting, `ukb(b)` — unfailing Knuth-Bendix completion;
`&, |, xor, ==>` — Boolean connectives "and", "or", exclusive "or", implication;
` ``=='' ` is the syntactic equality on terms; `=E=` is the equivalence relation on terms defined by the set `E` of equations contained in a calculus.
"Proof (of a statement) in initial" means (as standard) a proof of this statement for the initial model of the considered calculus (theory).

2 The Prover Principles and Design

Programming system and languages So far, we choose `Haskell` as implementation language. Here we shall call it IL, for generality. The prover is a program written in IL which processes *specification* data. A specification represents a *calculus* in the *object language* (OL) of many-sorted equational specification.
Representation of a proof goal This is a data `g :: Goal` of IL containing 1) a calculus `calc = goalCalculus g`, 2) the statement `f = goalFormula g` (in the predicate calculus language) which needs to be proved in `calc`, 3) the kind `mode = goalMode g` of the truth and proof: `InVariety` or `InInitial`.
Proof search state, representation of proof search
The prover has a set `StepKinds` of a few search step kinds (attempt kinds), each one presenting a particular method for an attempt to find a proof from the current search state (`Dumatel-1.06` has 5 main search step kinds). Each attempt is restricted by the resource `rcPerStep` measured in a certain conventional unit. When this resource is exhausted, the attempt stops, returning the current state.

The *search state* is represented by the IL data `ProofSearchState`. This is a tree which has sub-goals as nodes, and as edges it has search steps, where each attempt stores its current state. When the sub-goal is proved, it is replaced with the `true` node, and all the tree simplifies according to the meaning of each edge. The proof success is expressed by the tree of a single node containing the formula `true`. The current (large) action loop of the proof search is: choosing of an appropriate leaf in the search state tree and either continuing the attempt stored in this leaf or adding another (appropriate) attempts (with new edges) to this leaf. The new state is appended to the list of search states. So, the

intelligence of the proof search strategy depends only on how wisely it selects the current leaf, search step kind, and parameters for this step.

Each node in the search tree has the *kind*: `All` or `Any`. ``` ``All'' ``` means that the truth of the statement in this node is conjunction of the statement truth of each of the "sons" of this node. ``` ``Any'' ``` means disjunction of their truth. When a node is proved, the tree simplifies according to the kind of each node.

Prover This is the IL function `prove` which takes an initial search state and appends to it the list of the search states built according to the strategy. The default strategy for the proof search is: develop the state tree by *search in breadth.* The result list can be printed out. The printing formats allow the user to see the search progress with skipping details or in a more detail. The time being taken by each such an attempt is restricted by the given resource `rcPerStep`.

If the search fails, the result state list may unwind infinitely. Concerning this, we keep in mind that evaluation in `Haskell` is "lazy". Also it is always possible for a client function to apply the function `prove` and take first `n` states from its result.

The resource distribution approach Each method in a search step has the resource limit `rcPerStep`. Such a method calls for various sub-functions: completion procedure, trying substitutions with constants, recursive calls of the prover, and so on. Many of these sub-functions take an additional argument `rc` — a resource bound to be spent. Such a function also returns the remainder `rc'` of the resource. If this value occurs non-zero, the prover adds it when calls other sub-functions. The idea of this approach is to prevent the strategy from running into infinity in an unlucky search step. For example, the search step by completion may loop infinitely for some data, and it is impossible to uniformly predict when it will occur infinite.

Trace data for proof certificate Most of the prover functions take the *trace* data among the arguments and accumulate it in the result. For example, the function `reduce` returns the result term and also the trace sequence of the reduction: which current term is reduced to which term by applying which equation, etc. This is a provision for the proof certificate. Because an automaton can check the proof by applying one by one the elementary steps returned in the trace. Again, the "laziness" of Haskell is very useful here. Because if the client function does not use the trace, then the trace part of the result does not spend memory nor time.

Parts of a calculus A calculus consists of description of several *sorts, operators, variables, rules, equations*, BT (converted skolemized formulae), description of a term ordering ``` ``>>'' ``` and its operator *precedence.* We use here a "sugar" operator declaration which is not yet implemented. For example,

```
_+_ : Natural Natural -> Natural ...
```

means a binary infix operator on the sort `Natural`.

2.1 Rules, Equations, Term Ordering, Reduction

The list `rules` is an OL program. The interpreter `evaluate calc t` evaluates this program, contained in the `rules` part of the calculus `calc`, at the data term `t` as usual in rewriting programming, and with treating variables in `t` as constants. This evaluation is required to terminate.

A partial term ordering `''>>''` is an IL function to compare terms. It depends on the operator precedence table. It must satisfy the restrictions formulated in [Hsi:Ru]. The equation set in a calculus is often *not* Church-Rosser, and the prover does not rely on any particular order of applying equations. Instead, it exploits that unfailing completion is directed to a *ground Church-Rosser* equation set. Equations and rules must define the same equivalence relation `=E=` on terms. Often the initial equations appear as converted from the rules by re-orienting the rule sides according to the TRW ordering. The reduction by equations and equation superposition are subdued to the TRW ordering [Hsi:Ru]. The function `reduce calculus t`, reduces a term `t` to the normal form by *equations* under the given term comparison.

With equations, it is possible to do program computation as well as reasoning. Also the program evaluation can be modeled (at a cost overhead) by setting appropriate ordering and applying the function `completeAndReduce`. This method intermingles unfailing completion and "ordered" reduction.

2.2 Boolean Terms

BT represent skolemized predicate calculus formulae in the form of Zhegalkin polynomials `f`. They have the meaning of the Boolean equation `f = 0 (= false)`. We prefer to use BT, with special unification and superposition methods for BT, because the connectives `&` and `xor` have more properties than just being AC operators, and we like to use these properties in forming superpositions. For example, the cancellation law for `xor` holds.

In our system, BTs appear in the calculus as in the following example (in the Section 3). The formula `forall [X,Y] (X > Y ==> not (Y > X))` specified for the calculus `list` is converted to the equation `(not (X > Y) | not (Y > X)) + 1 = 0`, and then, to the BT `(X > Y)&(Y > X)` (a monomial). This conversion is based on the correspondence

`0 <-> false`, `1 <-> true`, `& <->` multiplication, `+ <-> xor`, `(+1) <-> not`.

A BT is a (commutative) sum of several different monomials. Each monomial is a (commutative) product of several different *atoms*. A monomial has an integer modulo 2 as its coefficient. The law `A & A = A` holds here. In the refutational proof, the prover applies the formula *negation*, *skolemization*, bringing to a *conjunctive normal form*. The disjuncts are converted further to BTs. The obtained BTs are added to the calculus, and there applies completion, with the aim to derive a BT `true`, which stands for the equation `true = false`.

2.3 Completion

Its function `ukbb rc calc goals <other arguments>` is designed after the principles by [Hsi:Ru] and also applies various optimizations. It is also extended to process Boolean equations in the form of BT. The method's ideology for the BT part is of the `RN+` strategy by [Hsi]. The procedure also applies intermediate reduction of `goals`. Completion stops when the resource is zero, **or** all the goal *facts* are reduced to trivial **or** the current set of the facts is *complete*. In returns a set of facts and the resource remainder `rc'`.

We also apply certain optimizations: with a special ordering on b-monomials, a stronger reduction relation on BT, and others.

2.4 The Search Step Kinds

They are
`(pCp)` positive goal completion and reduction (always pre-applied),
`(cnf)` bringing to conjunctive normal form (always pre-applied),
`(arC)` proof by substituting arbitrary constants for variables,
`(ec)` proof by parting equational conditions from implication,
`(nCp)` negation (, skolemization) and refutation by completion and cases,
`(ind)` induction by an expression value — for a universally quantified formula and the goal mode `InInitial`.
`(lsi)` proof by searching lemmata (LSI abbreviates "Lemma Search for Initial model").

`(pCp)` performs a limited number of completion steps, together with reduction of the formula. If the formula simplifies to `true`, then the goal is proved and deleted.

Induction by an expression value `(ind)`: when in a given calculus a sort `S` is attributed with the annotation `GeneratedBy <list of operators>`, (for constructing ground terms of this sort), the prover recognizes the correctness of proving a statement for `S` by induction by the sort construction with the given operators. The prover tries various expressions for induction by their value. In the current version, an expression for induction can be only a variable from the list under the `''forall''` construct in the goal formula.

We skip here explanation for the inference rules `arC, ec`. Let us describe shortly the remaining rules of `nCp` and `lsi`.

Refutation by completion and cases

The functions `refuteByCompletion, proveByNegationAndCompletion` implement the inference rule `(nCp)`. A calculus specification may contain a construct for finite enumeration of a sort with constants. And the above two refutation procedures rely on such construct. For example, the library calculus `bool` provides the enumeration `[true, false]` for the domain `Bool`. For expressing such enumeration, the prover has the construct

```
FiniteEnumeration S [c_1,...,c_n].
```

Its meaning is: only those models are taken in account for the proof, in which each value of the sort `S` coincides with some value listed in the enumeration. Respectively, the refutation applies completion together with equating the enumeration constants to the selected terms. Half of the resource is spent on the attempt by completion only. If this fails, the remainder is spent on completion with finding and applying the relevant cases. In each "case", completion applies to the calculus extended with the equations $g_i = c_i$, where c_i is one of the enumeration constants for the domain of a ground term g_i. To make the procedure more feasible, the part of `refuteByCompletion` applies certain heuristics for selection of appropriate terms g_i.
Lemma search This is a procedure of looking through the candidate formulae for lemma, with a certain fast check for rejection, and with the attempt of inductive proof (under a certain mean resource) for the candidates which have passed the check. The prover adds the proved lemmata to the calculi of all appropriate nodes in the current search state. This approach of LSI increases greatly the set of practically provable statements.

3 Proof Search Example.

Let us define in OL the calculus `list` for ordering lists. The first line of the below IL program builds the library calculus `boolCalc`, and the further lines add declarations to extend it with the needed sorts, operators, and equations. By all this, it imports the sort `Bool` together with the Boolean connectives and their laws (equations). This forms the calculus `list`. It has the sort `Elt` for list elements and sort `List` for lists over the domain `Elt`. The empty list operator `nil` and the operator `":" : Elt List -> List` for prepending an element are the constructors for the list data. The declaration `SortGen List [[nil, ":"]]` helps the prover to recognize that induction by these constructors for `List` is a correct way to prove statements for the initial model of the calculus `list` with respect to the domain `List`.

The operator `‘‘>’’` is for an order relation on `Elt`. The library function `addFormulae` adds to the calculus the formula expressing a couple of usual axioms for the properties of `‘‘>’’`. This predicate calculus formula is converted to BT and takes part in completion during the proof search. The predicate `eq_Elt` is for equality on the domain `Elt`. The predicate `isOrdered` is for expressing that a list is ordered with by the relation `‘‘>’’`.

The `Rules` part of this calculus actually contains the *program* to evaluate the ground terms constructed via the above operators. This is evaluation by rewriting, each rule applying "from left to right". For example, the library function call `evaluate list (insert b (a: c: nil))` results in the term `a: b: c: nil`.

3.1 A Digression: the Idea of Equational Reasoning for Programs

In order to reason about this program, the prover adds equations made from these rules — see in the sequel the call of the library function

`rulesFromCalculusToEquations`. The term ordering `cp` is set as a certain `rpos` library function (currently, the default one), which details we skip here. The library function `‘‘prove’’` does reasoning about the "program" of `list` in terms of the above *equations*, by applying them, maybe in both directions, by *superposing* them, and also comparing terms by `cp` to find which expression is "simpler". The equation set is considered rather as a *calculus* than a program for evaluation. For example, concerning the above program `insert`, the prover uses that the "input" term `insert b (a: c: nil)` *equals* to the "result" term `(a: b: c: nil)` modulo the equations obtained in the calculus `list`, and also that the latter term is conceptually "simpler" (by the TRW ordering `cp`) than the former.

Of course, this approach is applied to all programs. Also this approach is for reasoning about algebraic objects, in mathematics.

Concerning application to the program analysis, we stress that TRW and equational reasoning methods (as completion) really use all the information contained in a set of equations (the *completeness* property of the method).

3.2 Continuing with Example

The predicate `isOrdered` is defined in the rules via `‘‘>’’` and the auxiliary operator `isOrd`. The operator `insert` for inserting an element to a list according to the order `‘‘>’’`, and its auxiliary operator `ins`, are bound in mutual recursion.

```
boolCalc = bool_default rpos
list     = addFormulae preList
                    (forall [X,Y] (X eq_Elt Y  xor  X > Y   xor  Y > X))
 where
 preList =
  (\calc -> addEquations calc $ rulesFromCalculusToEquations calc) $
  addDeclarations_default boolCalc $
  Calculus
  {Sorts [Elt, List],   SortGen List [[nil, ":"]],
   Operators
   {nil   : List
    _:_   : Elt List -> Bool    (ParsePreceds 5 5),
    _>_   : Elt Elt -> Bool     ...,
    eq_Elt: Elt Elt -> Bool (...Commutative), --equality predicate on Elt
    insert    : Elt List -> List           ...,
    ins       : Elt Elt List Bool -> List  ...,
    isOrdered : List -> Bool               ...,
    isOrd     : Bool Bool -> Bool          ...,
    a, b, c   : Elt           -- constants for constructing list examples
   }
   opPrecedDecls = [... [insert, ins, isOrdered, isOrd, :, nil, >,
                         eq_Elt, a, b, c, false]], ...
   TermComparison = rpos
   Variables      = [X Y Z :  Elt,  Xs Ys Zs : List,  bo : Bool],
   Rules =
```

```
[X eq_Elt X -> true,                          -- laws for equality on Elt
 a eq_Elt b -> false,  a eq_Elt c -> false,  b eq_Elt c -> false,
 X > X -> false ,                             -- laws for order on Elt
 a > b -> false,     a > c -> false,
 b > a -> true,      b > c -> false,
 c > a -> true,      c > b -> true,

 isOrdered nil      -> true,
 isOrdered (X:nil)  -> true,
 isOrdered (X:Y:Ys) -> isOrd (X > Y) (isOrdered (Y:Ys)),
   isOrd true  bo -> false,
   isOrd false bo -> bo,

 insert X nil      -> X : nil,
 insert X (Y : Xs) -> ins X Y Xs (X > Y)
   ins X Y Xs true  ->  Y : (insert X Xs),
   ins X Y Xs false ->  X : Y : Xs         ]
```

The above declaration `opPrecedDecls = ...insert, ins, >, ...` defines the operator precedence relation `preced`. By setting the precedence the user gives the prover a notion of which "program" (operator) is simpler, and gives a certain direction of reasoning. Together with the library function `rpos`, it defines the term comparison related to this calculus. In particular, due to this precedence, the prover will consider the term `(insert a (b:Xs))` as more complex than `(ins a b Xs (a > b))`, so that the former will be replaced with the latter, and not the reverse.

Goal setting. Example

Prove that if a list `Xs` *is ordered, then the list* `(insert X Xs)` *is ordered.* This is actually an important part for verification of the program ``‘‘insert’’``. In our system, this means to prove the above statement in the initial model of the calculus `list`. The user IL program ``‘‘main’’`` is short. It parses the goal formula

```
forall [Xs, X] (isOrdered Xs ==> isOrdered (insert X Xs))
```

to `fF :: Formula` and builds the `Goal` expressing the problem of derivation

```
list  |-InInitial-  fF.
```

It makes the initial search state `initState` from this goal, and applies `prove rcPerStep initState` (for this example, it is sufficient to set `rcPerStep = 2*10^6`). It also prints out the result list of the proof search states.

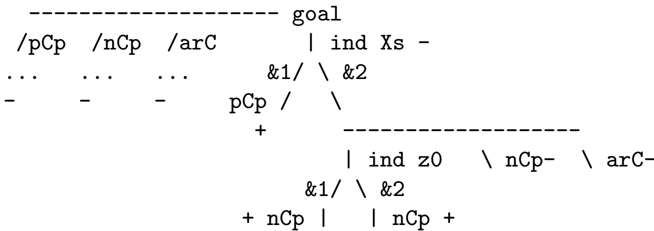
Here we skip the definition of the function ``‘‘main’’``. Let us describe how the prover forms the successive search states (trees).

First, the strategy applies all the following fitting *search step kinds* to the initial state: `pCp, nCp, arC, ind`. For this example, we skip a particular search step LSI, in order to demonstrate the main and regular part of the strategy. This stage produces the search tree of four leaves. The kind of the tree root is `Any`, because by the meaning of the prover standard search steps, the prover needs

to prove at least one of these leaves. In the next pass-through, the prover tries, in succession, to apply the corresponding proof methods in these leaves, trying to prove the current leaf under the resource `rcPerStep`. For the evident reason, this attempt fails for the leaves of `pCp, nCp, arC`, and the prover spends some resource for this.

The leaf of `ind Xs` means induction by the value of `Xs`. Further, this leaf branches to the base of induction (substitute `Xs = nil`) and inductive *step*. The goal formula of the "step" has new variable `z0`. The node of induction has the kind `All`, because by its meaning, both the base and "step" goals need to be proved.

The prover continues this *search in breadth* by visiting the current set of leaves, except the ones, which are skipped by various optimizations in the strategy and also by the user marks (hints) `''closed''`. After several steps, the prover forms the state tree shown schematically below. In this picture, the mark "-" near a node means that this node is not proved, so far, and "+" means that it is proved.



```
   -------------------- goal
  /pCp  /nCp  /arC       | ind Xs -
 ...    ...    ...     &1/ \ &2
 -      -      -    pCp /   \
                      +      -------------------
                              | ind z0   \ nCp-  \ arC-
                          &1/ \ &2
                     + nCp |   | nCp +
```

The symbols `&1` and `&2` denote the branches of the induction base and *step* respectively. The induction by `Xs` proves its base trivially. And the *step* formula is

```
forall [Y]
  ( (forall [z0,X] (isOrdered z0 ==> isOrdered (insert X z0)))  ==>
    (forall [z0,X] (isOrdered (Y:z0) ==> isOrdered (insert X (Y:z0)))) )
```

In the further induction by `z0`, the "base" formula (substitute `z0 := nil`) is reduced by the `list` calculus to

```
forall [Y,X] (isOrdered (ins X Y nil (X > Y))).
```

The attempt with `nCp` builds negation for this formula and produces the calculus

```
list  U  [isOrdered (ins A B nil (A > B)) = false],
```

with the indefinite Skolem constants `A, B : Elt`, aiming to derive a contradiction. It tries completion for this — under the resource `rcPerStep/2`. But this occurs not sufficient.

Then, it *extracts a useful information from the failed proof attempt.* To do this, it searches for appropriate ground subterms in this completion result which domains are provided (in the calculus specification) with a finite enumeration.

The calculus includes `bool`, and the latter specifies an enumeration for the sort `Bool`. The "case" procedure finds a ground subterm `A > B`, which value cases may simplify the search. The first case `A > B = true` is added to the calculus, and the formula is reduced to `isOrdered (B : (insert A nil)) = false`, and then, to `isOrd (B > A) true = false`. The BT part of the calculus contains a representation of the law `(X > Y & Y > X) = false`. It superposes with the "case" equation producing `B > A = false`. This derives the equation

```
isOrd false true = false,
```

and then, `true = false`, finishing the refutation for the case. The case of `A > B = false` is refuted in a similar way.
Nested selection of subterms for "cases"
Further, there are applied several search steps, and among them — induction by `z0`. Its "step" formula is

```
forall [y01,z01]
(forall [y0] ((forall [X] (isOrdered z01 ==> isOrdered insert X z01)) ==>
 (forall [X] (isOrdered (y0:z01) ==> (isOrdered ins X y0 z01 (X > y0)))))
 ==>
 forall [y0]
 ( (forall[X]...)==> (forall[X] ( isOrd (y0 > y01) (isOrdered (y01:z01))
                    ==> (isOrdered ins X y0 (y01:z01) (X > y0))    ) ) ) )
```

This formula is negated and skolemized, several equations are added to the calculus. Here refutation deals with the ground equations like

```
isOrdered (ins A y0 (y01:z01) (A > y0)) = false.
```

It considers the cases for the term `A > y0`. It fails to refute this set of two cases. Then, it searches among the facts produced by completion for new ground terms suitable to consider their cases. It finds a new subterm `A > y01`. Refutation by completion considers the sub-cases for this term: the procedure makes recursion. This continues until either the resource is out or the complete set of the cases is refuted. In our example, it finishes with the report of kind

```
Proof by negation and completion for the goal ...There were considered
6 cases for appropriate ground subterms ... The branch is proved.
```

By this, the lower two leaves of the current tree (on the picture) become proved, the whole tree simplifies according to the kind in each node, and the tree is converted to the trivially true one.

So, this goal is proved by combining the above standard proof attempts. The successful branch contains two induction edges, and also the final attempt `nCp` of the proof by negation and completion together with considering "cases".

The whole search process is similar to human reasoning, when a human searches for the proof of the above statement for the program `''insert''`.

4 Possible Development Directions

There are many ways in which the current prover should progress. Let us name the three of them.

1. Similarly as with human reasoning in solving problems, really efficient methods are feasible only for a specialized subject domain. For example, sorting methods, finite groups, polynomials, and so on. This leads to specialized knowledge bases, and needs an interface to a CA library.
2. Various improvements and extensions are needed for the existing strategy. Many optimizations are possible for the BT processing and AC-Id completion.
3. It is useful to extend the object language with conditional rewriting, high-order operators, functoriality.

References

[Bu:Al] Buchberger, B., Dupré, C., Jebelean, T., Kriftner, F., Nakagawa, K., Văsaru, D., Windsteiger, W. *The THEOREMA Project: A Progress Report.* In: Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning), (98–113) August 6-7, 2000, St. Andrews, Scotland, A.K. `http://www.risc.uni-linz.ac.at/people/buchberg/main_publications.html`

[Bu2] Buchberger, B. *Algorithm-Supported Mathematical Theory Explanation: A Personal View and Strategy.* In Proceedings of AISC 2004 7th International Conference on Artificial Intelligence and Symbolic Computation, September, 2004, RISC Institute, Linz, Austria. Lecture Notes in Artificial Intelligence, Volume 3249, 2004, pages 236 – 250.

[Hsi] Hsiang, J. *Refutational theorem proving using term-rewriting systems.* Artificial Intelligence, 1985, Volume 25, pages 255-300.

[Hsi:Ru] Hsiang, J., Rusinowitch, M. *On word problems in equational theories.* In Th. Ottman (ed.), Proceedings of the Fourteenth International Conference on Automata, Languages and Programming, Karlsruhe, West Germany, July 1987, Springer Verlag, Lecture Notes in Computer Science **267**, pages 54 – 71, 1987.

[Hu:Op] Huet, G., Oppen, D. *Equations and rewrite rules. A Survey.* In "Formal languages: perspectives an open problems", pages 349 – 405. New York, Pergamon Press, 1980.

[K:B] Knuth, D., Bendix, P. *Simple word problems in universal algebras.* In John Leech, editor, "Computational Problems in Abstract Algebra", (263–297), Pergamos Press, 1970.

[Lo:Hi] Löchner, B., Hillenbrand T. *A Phytography of Waldmeister.* AC Communications **(15)** (2,3) (2002) (127–133).

[Me1] Mechveliani, S. *Computer algebra with Haskell: applying functional – categorial – "lazy" programming.* In Proceedings of International Workshop CAAP-2001, pages 203–211, Dubna, Russia. `http://ca-d.jinr.ru/confs/CAAP/Final/proceedings/proceed.ps`

[Me2] Mechveliani, S. The `Dumatel` program system and book (manuscript). A preliminary version of 1.06-pre3 (half of the manual need update). `http://www.botik.ru/pub/local/Mechveliani/dumatel/1.06-pre3/`

[Sti] Stickel, M.E. *A Unification Algorithm for Associative-Commutative Functions*, Journal of the Association for Computer Machinery, Volume 28, No 3, (1981), pages 423–434.