Inductive Theorem Proving for Algebraic Specifications

- TIP System User's Manual -

Ulrich Fraus

Bavarian Research Center for Knowledge Based Systems (FORWISS)
University of Passau, Innstr. 33, D - 94032 Passau, Germany
E-mail: fraus@forwiss.uni-passau.de

Abstract

This manual introduces an inductive theorem prover called TIP system (**T**erm **I**nduction **P**rover). This prover can be used to verify universally quantified systems of conditional equations over algebraic specifications. The proofs can be done in either an interactive or an automatic way. The reader learns how to use the TIP system, especially how to do inductive proofs in an interactive way. Besides, the internal proof algorithm and the built-in heuristics are explained and some useful techniques for successful theorem proving are presented.

Preface

This report refers to the TIP system version 3.0. Older versions may not have all the features described in this manual or may behave in a slightly different way than described in this manual. For more information on the distribution of the TIP system, error reports, suggestions, etc. please send electronic mail to fraus@forwiss.uni-passau.de. If you would like to get and install the prover on your machine(s) then please see appendix B.

Thanks to all the students who developed some parts of the code of the TIP system and especially to Heinrich Hußmann and Alfons Geser for supporting the development since its beginning in 1989.

Contents

1 Introduction

Inductive reasoning is a widespread method for proving universally quantified formulae over formal specifications. Such specifications contain the declaration of sorts (like natural numbers or lists) and function symbols (like multiplication or sorting of lists). To give some "meaning" to those function symbols a specification moreover contains a set of axioms. From a general point of view formal specifications can be seen as the description of the behavior of (mathematical) functions, software code or technical systems. Theorems are, possibly conditional, equations which state "new" information about the specified functions. Therefore, theorems can be seen as descriptions of special properties which should be fulfilled by the specified piece of software.

Of course one would like to know if such a property (theorem) is true for a given specification. Therefore one needs to prove that the theorem "follows" (is derivable) from the axioms of the specification independently of the values of the variables. To do such a proof all the variables appearing in a theorem have to be universally quantified. A theorem could look like:

$$\forall n \in \mathbf{N} : \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

This states the well-known formula of Gauß for calculating the sum of the first n natural numbers. A listing of the automatic proof of this theorem follows in section 1.3.

In most of the cases induction can be used to prove universally quantified theorems, especially if there are infinitely many possible values for some variables of the theorem. Examples would be again the natural numbers or the lists. In those cases a theorem cannot be verified by a finite case analysis over all possible values. So, some scheme of induction has to be used, i.e. one tries to transform every non-trivial case such that an induction hypothesis can be applied. The main task of systems for inductive theorem proving is now to decide which cases are "easier" and to find a way of reducing the non-trivial cases to "easier" ones. Of course this is possible only if there exists some kind of iterative scheme for all possible values.

Also the tasks of generalization (replacing expressions by variables) and the usage of lemmas[1] are necessary for non-trivial theorem proving.

Theorems may be either unconditional or conditional and they may use variables of any sort, i.e.:

$$\forall n, m, k \in \mathbb{N}: \; n^m * n^k = n^{m+k} \qquad \text{or} \qquad \forall x, \forall y: (x \neq 0) \;\Rightarrow\; f(x, y) = g(x, y)$$

The first example states an unconditional mathematical formula. Of course such mathematical theorems can be proved by induction. They can be seen as one special case of inductive theorem proving in general, because you always can distinguish the two cases *0* and *n+1*. The second example is a conditional equation and it means that two functions f and g (from a given specification) behave in the same way for all possible values of x and y unless x is zero. This kind of problem is undecidable in general but often induction is powerful enough to solve it, even if there are infinite many possible values for x and y.

1.1 Overview

The theorem prover called TIP System (**T**erm **I**nduction **P**rover) which is introduced in this work is a tool for proving the correctness of conditional theorems. These theorems together with the specification on which they are based are the input of the system. Then the proofs can be done either automatically or interactive. The output of the prover is the information if a theorem could be proved or disproved. Moreover, if a theorem has been proved then the TIP system provides a pretty-printed listing of this proof. Another system which has quite the same behavior but a different proof algorithm is the well-known Boyer-Moore prover [BM79]. One of the main differences between these two systems is the

1. In this paper a lemma is defined to be a theorem which is used for the proof of another theorem.

possibility of TIP to deal with user interaction. The amount of this interaction can be selected by setting several system parameters. Especially for non-trivial proofs this feature is an advantage for the user of the TIP system.

Other important properties of TIP are the powerful handling of conditions and its possibility to search for a proof in different ways simultaneously. A study in 1992 showed that there are theorems where TIP is superior to Boyer-Moore as well as the other way round.

The development of the TIP System started in 1988 at the University of Passau. The first version of this inductive theorem prover has been implemented by Ulrich Fraus during his diploma thesis at the University of Passau. It was available in summer 1989. This version was mainly based on the RAP system, a rapid prototyping environment for algebraic specifications. The RAP system has been developed by Heinrich Hußmann [GH85]. Most of the basic data structures, the general structure of the specification language and the command language have been adopted from the RAP system. The main difference between TIP and RAP is, that TIP tries to solve universally quantified equations while RAP tries to find solutions for existentially quantified equations.

Further extensions and improvements have been implemented by Ulrich Fraus with the help of Gregor Bittmann, Rainer Große-Gehling, Thorbjörn Hansen, Alexander Knapp, Thorsten Mieze, Klaus Rieger, Hubert Waibel and others. The main improvements since version 1.0 are conditional equations, generalization, AC unification, completion based proving, better proof listings and more intelligent proof heuristics. The current version 3.0 has been released in autumn 1992. The TIP system is implemented in (Standard-)PASCAL with some I/O routines written in ANSI-C. It consists of 36 modules with about 35.000 lines of code. The system is currently available for the operating system UNIX on SPARCstations.

The version 2.3 (without AC unification and conditional theorems) is also available for the operating system DOS on PC. This implementation is a little bit slow and the memory normally is not large enough for bigger proofs, but it is useful for testing the behavior of the prover and to run most of the standard examples.

If you need information on how to get the TIP system or if you want an update for your old version then contact Ulrich Fraus by electronic mail (fraus@forwiss.uni-passau.de) or postal mail. If you need information on installing the TIP system at your local computer(s) then see appendix B.

A detailed discussion of the theoretical basis of the TIP system together with a comparison to other existing theorem provers will follow in 1994 as dissertation of the author.

This manual is organized as follows: The rest of this chapter gives an introductory example for the use of TIP. We define the syntax of the TIP specification language formally in chapter 2 and explain the TIP command language and the user interface of the TIP system in chapter 3. All the topics related to the proof of theorems can be found in chapter 4. In the following chapter the editor for theorems is explained.

Some special features are discussed in chapter 6 and a semi formal description of the most important algorithms of TIP can be found in the following chapter. Finally a collection of ideas for further developments is summarized in chapter 7.

In the appendix a glossary, information about installing the TIP system, about the standard types and about already proved examples can be found.

1.2 Specifications and Theorems

The TIP system is a theorem prover which supports the verification of conditional or unconditional theorems by induction. These theorems have to be based on modular algebraic specifications. For this purpose the small but powerful specification language (defined for the RAP system [Hu85]) is used (see chapter 2). This language is based on the concept of hierarchical algebraic specifications ([Wi90]). The TIP system does a rigorous syntax check of the specification text. Besides, some checks for interesting properties are performed (see section 6.2). These properties are the completeness[1] and termination[2] of

1. A sufficient criterion for the completeness w.r.t. the constructor functions for all argument sorts is tested.
2. A sufficient criterion for the termination (i.e., no infinite axiom applications) is tested with the lexicographic path ordering.

the axioms.

For the dialog with the TIP system a quite understandable command language (without unintelligible abbreviations) is implemented (see chapter 3). The central behavior of TIP is a loop over the command parser (for user input) and the command interpreter until the system is terminated by the user.

The activities of TIP after starting can be listed as follows:

- Evaluate the flags of the UNIX command line (if existing).
- Load the default types `BOOL` and `NAT` into the system.
- Parse all input files which are given as arguments in the UNIX command line.
- Do the completeness and termination checks for all user defined types.
- Build the rule-database from the axioms of all types.
- Check if a personal initialization file `.tiprc` exists in the user's home directory and execute it if so.
- Repeat the following activities until the user terminates the system:
  - Use the command parser to get a correct command from the user.
  - Perform all the actions which are associated with this command.

The central proof algorithm is invoked by entering the **PROVE** command. This causes the system to change to a special mode for proving theorems (see chapter 4). In this PROVE mode there again is the above kind of command loop. The only difference is that if this nested loop is terminated then TIP returns to the main command loop instead of terminating itself. Two other modes of this kind are editing theorems (see chapter 5) and on-line help for the TIP commands (see section 3.3).

Of course, the main task of the TIP system is proving the validity of theorems by induction. It is possible to do the inductive proofs either in an automatic or an interactive way. This can be selected by adjusting the corresponding system parameters (see section 4.5). Thereby it is determined which applications of proof steps have to be decided by the user and which ones are done in an automatic way. For example the boring work of transforming a term into its normal form (*rewriting*) normally is done by the system and the "interesting" decisions are left to the user.

Possible proof steps in TIP are: Rewriting, case analysis, application of an induction hypothesis, generalization and application of a lemma. The case analysis can be done over one or several variables or over a new term which is added to the conditions.

If a theorem is proved to be valid by the TIP system then this means, that every constructor ground substitution[1] is derivable from the axioms. In model theory this is called "valid in the initial constructor-generated model of the specification".

1.3 A Simple Example

In this section, we give a first example what a specification and a theorem can look like in the TIP system. Besides, a listing of an interactive proof session is given. We assume that a specification has been written into a UNIX text file named '`gauss.tip`'. This input file is read by the TIP system. Let `gauss.tip` contain the following TIP specification text which is to be understood as a simple specification of finite sums of natural numbers together with a theorem stating the formula of Gauß:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Before reading this input file `gauss.tip`, TIP loads a couple of basic standard specifications, e.g. a specification `NAT` of the natural numbers. The specification in `gauss.tip` is based on the natural numbers and therefore we start with a shortened version of NAT. We use dots (...) for those parts of the specification which are not interesting for our example. (These dots are not part of the TIP syntax!) For the complete text of `NAT` see appendix D.

In the following specification all keywords are printed boldface to make the structure more obvious:

1. All variables of the theorem are replaced by terms built only from constructor functions.

```
type NAT

 basedon BOOL
 sort Nat

 cons zero : Nat,
      succ : (Nat)Nat

 func add : (Nat,Nat)Nat,
      mult : (Nat,Nat)Nat,
      …

axioms all (x,y,z,r : Nat)

 add(x,zero) -> x,
 add(x,succ(y)) -> succ(add(x,y)),
 add(zero,x) -> x,
 add(succ(x),y) -> succ(add(x,y)),
 …
 mult(x,zero) -> zero,
 mult(x,succ(y)) -> add(mult(x,y),x),
 …

endoftype

noetherian NAT
complete NAT
```

The **type** NAT defines the set of the natural numbers. It is hierarchically based on another standard type BOOL (keyword **basedon**), but this type is not important for our example. NAT introduces a **sort** Nat to denote the set $\mathbf{N}$, a constant function zero to denote 0, and an unary function succ for the successor function.

Both functions, zero and succ, are the constructor functions (keyword **cons**) of the sort Nat. The idea of constructor functions is that every object of a sort can be constructed by such functions, e.g. every natural number *n* can be viewed as a n-th successor of 0. Proceeding on this assumption the TIP system performs case analysis over a complete set of constructor terms (see section 4.3.2).

For the purpose of our example we need only two further functions on numbers: add and mult are intended to compute the sum or the product of two natural numbers, respectively. Their signatures are given following the keyword **func**.

The **axioms** specify the behavior of add and mult by means of a system of term rewrite rules. To indicate this difference to equational logic, the axioms are written with an arrow instead of the equality sign. The variables are prefixed by the keyword **all**. The term rewriting relation generated by NAT is, in addition, noetherian and complete. The lines **noetherian** NAT and **complete** NAT state these properties.

Now the specification SUM_NAT can be based on (keyword **basedon**) this description of the natural numbers:

```
type SUM_NAT
 basedon NAT

 func sum : (Nat)Nat, {sum from zero to n}
      div2 : (Nat)Nat {number divided by two}

axioms all (n : Nat)

 sum(zero) -> zero,
 sum(succ(n)) -> succ(add(n,sum(n))),

 div2(zero) -> zero,
 div2(succ(zero)) -> zero,
 div2(succ(succ(n))) -> succ(div2(n)),

endoftype
```

```
theorem GAUSS
 basedon SUM_NAT,NAT

 forall n: Nat

 goals
   sum(n) = div2(mult( n,succ(n) ))
endoftheorem
```

An input text of TIP is composed from **type** declarations and **theorem** declarations. A type defines data domains (sorts) and function symbols on them in an algebraic way. A theorem consists essentially of a system of (conditional or unconditional) equations to be proved by the system.

The type `SUM_NAT` defines the functions `sum` and `div2` which have the informal meaning that `sum(n)` denotes $\sum_{i=0}^{n} i$ while `div2(n)` means $\lfloor n/2 \rfloor$, i.e. *n* divided by two and then rounded down. `SUM_NAT` has to be hierarchically based on the standard type `NAT`, because the natural numbers and the basic function `add` for addition are used.

The end of a type or a theorem is indicated by **endoftype** or **endoftheorem** respectively.

The **theorem** `GAUSS` states that the sum of the first `n` natural numbers is equal to the half of `n` times `n+1`. It is intended to verify this theorem by induction with the TIP system. All variables which are used in the theorem are universally quantified (keyword **forall**). The **goals** part of a theorem is a system of (conditional or unconditional) equations.

A more detailed description of the syntax for the input of the TIP system is given in chapter 2.

An example session with the TIP system

When the TIP system is invoked[1], given the input file `gauss.tip` described above as an argument, it issues the following dialog on the screen (if your terminal is able to print boldface and underlined text).:

```
TIP System Version 3.0 SUN 4.1/UNIX

Reading standard type declarations.

STEP 1: Syntax check
Reading type SUM_NAT.
Reading theorem GAUSS.
No syntax errors detected.
```

Before this step TIP quietly reads the standard types BOOL and NAT. Then input file is parsed and all types and theorems which have been found are reported.

```
STEP 2: Property check
noetherian SUM_NAT -- by criteria
complete SUM_NAT -- by criteria

STEP 3: Interactive induction prover
Preprocessing the axioms.

TIP>
```

At this point TIP has checked the syntax of the input file and has tested the completeness and the termination of the axioms of the type `SUM_NAT` (see section 6.2 for details about these tests). Now the system is ready for executing commands. This is signaled by printing the prompt of the STANDARD mode ("**TIP>**"). Below a short dialogue with the system is shown. The user input is

1. Normally the system is started by the command "**TIP <file name(s)>**" or "**tip <file name(s)>**" from a UNIX shell. <file name(s)> denotes the name(s) of the UNIX file(s) containing the input text.

printed *italic*.

To get an overview of the types and theorems known within the current session we can use the command *dir* (for 'directory'):

```
TIP> dir
Types:
BOOL                      NAT                       SUM_NAT

Theorems:
GAUSS

TIP>
```

Remember that the directory contains besides the types from the input file the two standard types BOOL and NAT. The *list* command lists up the text of a type or theorem definition:

```
TIP> list GAUSS

 theorem GAUSS

   basedon SUM_NAT,NAT

   forall n: Nat

 goals

   sum(n) = div2(mult(succ(n),n))

 endoftheorem

TIP> prove GAUSS
# CURRENT THEOREM: NO. 1
sum(X1) = div2(mult(succ(X1),X1))

TIP/PROVE>
```

The algorithm for proving theorems is started with the command *prove*. We want to find an induction proof for the given theorem. Thus we enter the PROVE mode which is indicated by the new prompt **TIP/PROVE>**. In this mode we can use some special commands for controlling and observing the current proof session. For instance, a fully automatic proof is forced by the command *go*. For this the system parameters should be set for automatic proving. These settings are the default values of the TIP system.

```
TIP/PROVE> go
# THEOREM NO. 1 PROVED BY SYSTEM.

TIP/PROVE>
```

The system tries to find a proof for the theorem GAUSS. The message "THEOREM NO. 1 PROVED BY SYSTEM" shows us that it was successful. Now we want to see how TIP did this proof. To get a nice listing the command *show proof* is used. Notice that we are still inside the PROVE mode. When we leave this mode all the information about the current proof is deleted!
In the following proof listing all variables are represented by the character 'X' followed by a number. This is necessary to allow the system to generate an arbitrary number of "new" variables during its search for a proof. At the right end of a line the rule or proof step which was used to derive the equation in this line is stated. If not enough place is left in this line then this information is printed in the line below. For example the term 'SUM_NAT.3' means that the third axiom of the type SUM_NAT was used for deriving the equation in this line. You can display this rule with the command *show rule SUM_NAT.3*.

```
TIP/PROOF> show rule SUM_NAT.3
Rule SUM_NAT.3 (noetherian):
 div2(zero) -> zero

TIP/PROVE> show proof

TIP System Version 3.1 SUN 4.1/UNIX                          Listing of Proof

PROOF OF THEOREM

 sum(X1) = div2(mult(succ(X1),X1))
 sum(X1) = div2(add(mult(X1,X1),X1))                              (by NAT.13)


 COMPLETE CASE ANALYSIS on X1                                          => [1]
 (Generated from the rules for function sum)

 1. CASE: X1 = zero

   - sum(zero) = div2(add(mult(zero,zero),zero))
   - 0 = div2(0)                                                (by SUM_NAT.1)
   - 0 = 0                                                      (by SUM_NAT.3)
   CASE 1. finished.

 2. CASE: X1 = succ(X2)

   - sum(succ(X2)) = div2(add(mult(succ(X2),succ(X2)),succ(X2)))
   - succ(add(X2,sum(X2))) = div2(add(mult(succ(X2),succ(X2)),succ(X2)))
                                                                (by SUM_NAT.2)
   - succ(add(X2,sum(X2))) =
       div2(add(add(mult(succ(X2),X2),succ(X2)),succ(X2)))        (by NAT.11)
   - succ(add(X2,sum(X2))) =
       div2(add(add(add(mult(X2,X2),X2),succ(X2)),succ(X2)))      (by NAT.13)
   - succ(add(X2,sum(X2))) =
       div2(add(succ(add(add(mult(X2,X2),X2),X2)),succ(X2)))       (by NAT.4)
   - succ(add(X2,sum(X2))) =
       div2(succ(add(succ(add(add(mult(X2,X2),X2),X2)),X2)))       (by NAT.4)
   - succ(add(X2,sum(X2))) =
       div2(succ(succ(add(add(add(mult(X2,X2),X2),X2),X2))))       (by NAT.6)
   - succ(add(X2,sum(X2))) = succ(div2(add(add(add(mult(X2,X2),X2),X2),X2)))
                                                                (by SUM_NAT.5)
   - add(X2,sum(X2)) = div2(add(add(add(mult(X2,X2),X2),X2),X2))  (succ removed)
   - add(X2,div2(add(mult(X2,X2),X2))) =
       div2(add(add(add(mult(X2,X2),X2),X2),X2))               (by IND.HYP.[1])
   - add(X2,div2(X26)) = div2(add(add(X26,X2),X2))          (by GENERALIZATION)


    COMPLETE CASE ANALYSIS on X2                                       => [2]
    (Generated from the rules for function add)

   2. 1. CASE: X2 = zero

      - add(zero,div2(X26)) = div2(add(add(X26,zero),zero))
      - div2(X26) = div2(add(add(X26,0),0))                        (by NAT.5)
      - div2(X26) = div2(add(X26,0))                               (by NAT.3)
      - div2(X26) = div2(X26)                                      (by NAT.3)
      CASE 2.1. finished.

   2. 2. CASE: X2 = succ(X98)

      - add(succ(X98),div2(X26)) = div2(add(add(X26,succ(X98)),succ(X98)))
      - succ(add(X98,div2(X26))) = div2(add(add(X26,succ(X98)),succ(X98)))
                                                                   (by NAT.6)
      - succ(add(X98,div2(X26))) = div2(add(succ(add(X26,X98)),succ(X98)))
                                                                   (by NAT.4)
      - succ(add(X98,div2(X26))) = div2(succ(add(succ(add(X26,X98)),X98)))
                                                                   (by NAT.4)
      - succ(add(X98,div2(X26))) = div2(succ(succ(add(add(X26,X98),X98))))
                                                                   (by NAT.6)
      - succ(add(X98,div2(X26))) = succ(div2(add(add(X26,X98),X98)))
                                                                (by SUM_NAT.5)
      - add(X98,div2(X26)) = div2(add(add(X26,X98),X98))          (succ removed)
      - div2(add(add(X26,X98),X98)) = div2(add(add(X26,X98),X98))
                                                               (by IND.HYP.[2])
      CASE 2.2. finished.

THEOREM PROVED
```

```
STATISTICS                                THEOREM
  Lemma applications          :              0
  Generalizations             :              1
  Induction hypotheses used   :              2
  Number of case analyses     :              2
  Total number of cases       :              4
  Decomposition steps         :              2
  Rewrite rule applications   :             18
  CPU-time used (in msec)     :           6033

TIP/PROVE>
```

In the listing above TIP shows how the proof of the formula of Gauß was done. At the right border of our proof output the system lists the rules which have been used to get the equation in the according line. The important steps like case analyses, generalizations and applications of hypotheses are printed boldface. Each case is finished if the terms on both sides of the last equation are equal. The generation of an induction hypothesis is shown as "=> [1]" in the listing. Also some indent is used to give an impression of the tree structure inside of this proof.

TIP can generate a pretty-printed listing of a proof, which allows other people to read and understand the proof. This form of output is similar to the Boyer-Moore prover, but very different from the completion based proof systems (see section 6.4), because these provers subsequently add and remove new rules. Therefore the output of 'natural induction' provers is much easier to understand for non-experts.

For the formula of Gauß a schematic representation of the proof tree (without the simplifying rewrite steps) would look like represented in figure 1. Up to now, such trees cannot be generated by the TIP system.

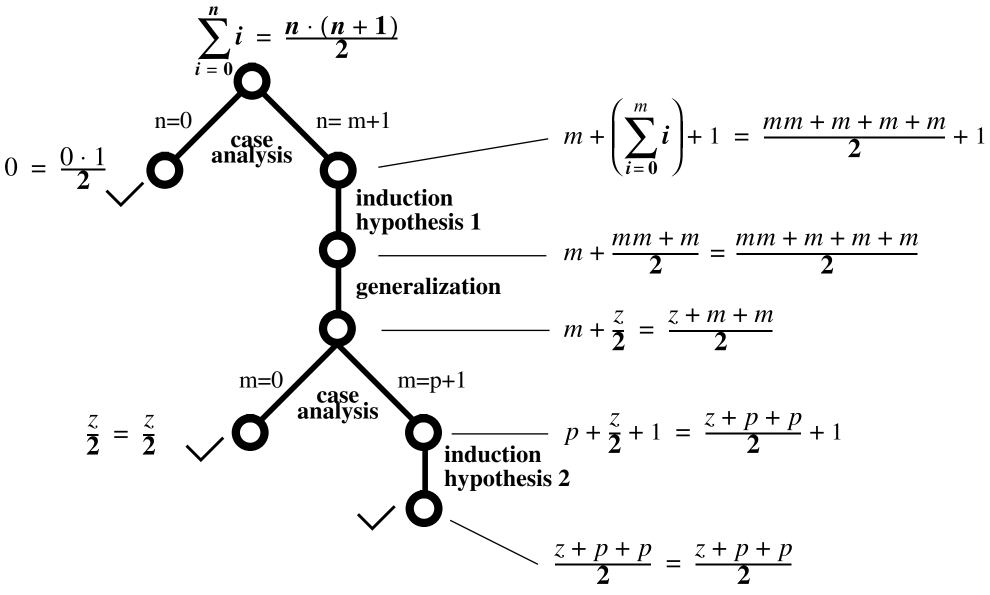


Fig. 1: The structure of the proof tree of the formula of Gauß

The statistics at the end of the proof listing show which proof steps have been used how often and how much CPU time it took to calculate this proof. This is useful to see which "kind" of proof was found.

The `exit` command is used to leave the PROVE mode and, once more, to leave the TIP system itself.

```
TIP/PROVE> exit
```

```
TIP> exit
```

For further information about the command language see chapter 3. Details about the PROVE mode and the interactive way of theorem proving follow in chapter 4. If you are not familiar with the terminology of rewriting and theorem proving then you should read the glossary in appendix A.

Another method of proving inductive theorems is the completion-based induction. A simple version of this proof scheme is also implemented in the TIP system. Its principles and handling are explained in section 6.4. This method is also called Knuth-Bendix completion or "inductionless" induction.

1.4 Where to Find More Examples

Some larger examples can be found in appendix E. These examples show the interactive process of proving as well as some more interesting proof jobs. You can see where and why the use of lemmas is necessary and how to deal with conditional theorems. The topics of those examples are:

- '*Double Reversion of a Sequence*' (see appendix E.1 on page 96) shows the application of a special proof step called 'twin step',
- '*Commutativity of Addition on*' (see appendix E.2 on page 97) shows nested inductions,
- '*Even or Odd*' (see appendix E.3 on page 99) shows the proof of a conditional theorem,
- '*Next Higher Even Number*' (see appendix E.4 on page 100) shows the need for doing a case analysis in the conditions,
- '*Binomial Coefficient*' (see appendix E.5 on page 104) shows a theorem where interactive rewriting is necessary,
- '*The "if" Function*' (see appendix E.6 on page 105) shows an interactive proof session.
- '*Binary Sorting*' (see appendix E.7 on page 109) shows an interesting generalization and the use of lemmas in the TIP system.

In the literature (see References on page 87) you can find examples for TIP in [FH92]. Larger experiments with specifications are described in [Ge86] (the specification of a microprocessor), in [HR87] (a compiler for an applicative language) and in [Pa87] (the specification of a programming language for concurrent systems). A summary of the experiences with earlier versions of TIP can be found in [FH92].

If you have installed the TIP system on your computer correctly (see appendix B) you should have a directory called "examples". There you can find several specifications and theorems. We have made an effort to insert enough comments into every specification and theorem to make it understandable. Some of these examples have been taken from [GL91], the appendix of [BM79] and other papers about inductive proving.

2 TIP Specification Language

To start the TIP system you have to give the name of the input file as an argument to the command. This input file has to contain specifications (types) and theorems written using the TIP specification language. It is also possible to put comments and property information about the types inside this input file. In this chapter you learn how to build up the specifications and the theorems which should be used.

2.1 General Structure

The specification language of the TIP system consists of a hierarchy of (algebraic, formal) **types** together with a number of so-called **theorems**. A type (specification) defines a signature (i.e. a set of sort-identifiers and a set of function-identifiers with domain and codomain) together with a set of positive conditional axioms. A theorem consists of a system of (conditional) equations together with a declaration of universally quantified variables. Types are built up in a modular way: a type may make use of (i.e. be "based on") another type. Similarly, theorems are defined based on a set of types. In the following the context-free syntax of the TIP specification language is defined using an extended Backus-Naur notation. The context conditions and the semantics of a specification are given in an informal style. The following sections are mainly based on [Hu85].

2.2 Conventions for the Syntax Description

We use an extended Backus-Naur form for the description. Nonterminals are enclosed in angular brackets (<…>) and terminals are written boldface. Concatenation of strings is denoted by juxtaposition. Further meta-symbols of the syntax description are

{ } }* |

The string *A* | *B* denotes the choice between the strings *A* and *B*, { *A* } the choice between *A* and the empty string, { *A* }* the repetition of *A* an arbitrary number of times.

We distinguish between the lexical and the context-free part of the syntax in order to define the role of delimiter symbols.

2.3 Lexical Elements

Lexical elements are the basic parts of a specification text. There are 5 classes of lexical elements: identifiers, keywords, numbers, special symbols and display-formats. A display-format is a special kind of string used for the definition of mixfix operations. The syntax is as follows:

<id> ::= <letter> { <letter> | _ | <digit> }*

<keyword> ::= **ac_info** | **all** | **assumed** | **axioms** | **basedon** | **cons** | **complete** | **display** | **endoftask** | **endoftheorem** | **endoftype** | **except** | **exist** | **forall** | **func** | **goals** | **hidden** | **kb_weight** | **noetherian** | **pr_ordering** | **sort** | **task** | **theorem** | **type** | **unknown**

<number> ::= <digit> { <digit> }*

<special symbol> ::= **:** | **(** | **)** | **,** | **=** | **=>** | **->** | **&**

<display-format> ::= "{ <display-substring> { _ <display-substring> }* }"

<letter> ::= **A** | **B** | **C** | … | **Z** | **a** | **b** | **c** | … | **z**

<digit> ::= **0** | **1** | **2** | … | **9**

The following additional rules hold for the lexical elements:

- A <keyword> is not allowed as an <id>.
- A <keyword> has to be written lowercase.
- A <number> has to consist of less than 10 digits.
- Two identifiers are considered to be equal if they coincide in the first 24 characters. Upper- and lower-case characters are distinguished, for example the identifier '`Nat`' is not equal to the identifier

'NAT'.

- As a <display-substring> every sequence of printable ASCII characters is allowed which does not contain the characters '"' (double-quote) and '_' (underscore). A display-substring must not exceed 24 characters. See section 10 of this chapter for details on the use of display-formats.

Subsequent lexical elements are delimited by strings consisting of one or more of the following parts (so-called *delimiters*):

- blank, line feed, horizontal tabulator
- comments.

Delimiters which stand before or after a special symbol may be omitted. The delimiter rules do not apply to display-substrings. The delimiter characters above may appear within a display-substring as ordinary characters.

The keywords `task`*,* `unknown`*,* `exist` *and* `endoftask` *are not meaningful for the TIP system. These are just kept as keywords to qualify the parser to deal with original RAP specifications (which may include tasks) without unnecessary error messages. Thus if you have specifications which include tasks you can use them as input for TIP without removing these tasks.*

Any text not containing the character '}' is admitted as a *comment*. Comments have to be enclosed in braces '{' and '}'. Comments may be inserted between lexical elements without changing the meaning of the text.

2.4 Context-free Syntax

A TIP specification text has the following form:

<system>	::=	<unit> { <unit> }*
<unit>	::=	<type> | <theorem> | <task> | <property>

The specification text has to be composed from units which may be

- **types** i.e. algebraic specification modules, for defining abstract data types,
- **theorems** defining claims about the specified functions,
- **tasks** defining questions about the behavior of the specified functions, and
- **properties** of the specifications, to be assumed by TIP, or a mixfix representation for each function symbol.

2.4.1 Type Syntax

<type>	::=	**type** <id> { <primitives> } { <signature> } { <axioms> } **endoftype**
<primitives>	::=	**basedon** <id-list>
<id-list>	::=	<id> { **,** <id> }*
<signature>	::=	{ <sort-decl> | <funct-decl> | <visib.-restr> }*
<sort-decl>	::=	**sort** <id-list>
<funct-decl>	::=	**func** <functity> { **,** <functity> }* | **cons** <functity> { **,** <functity> }*
<functity>	::=	<id-list> **:** <args> <id>
<args>	::=	{ (<id-list>) }
<visib.-restr>	::=	**hidden** <id-list>
<axioms>	::=	**axioms** { **all** (<vars>) } <axiom> { **,** <axiom> }*
<vars>	::=	<var-decl> { **,** <var-decl> }*
<var-decl>	::=	<id-list> **:** <id>

<axiom> ::= { (<id>) } { <premises> } <rewrite>

<premises> ::= <equation> { **&** <equation> }* **=>**

<equation> ::= <term> { **=** <term> }

<rewrite> ::= <term> { **->** <term> }

<term> ::= <number> | <id> | <application>

<application> ::= <id> { (<term> { **,** <term> }*) }

2.4.2 Theorem Syntax

<theorem> ::= **theorem** <id>
{ <primitives> }
{ <forall-decl> }
{ <goals> }
endoftheorem

<forall-decl> ::= **forall** <vars>

<goals> ::= **goals** <cond-equation> { **,** <cond-equation> }*

<cond-equation> ::= { <premises> } <equation>

2.4.3 Property Syntax

<property> ::= <noetherian> | <complete> | <acinfo> | <kbweight> | <pr-ordering> | <display>

<noetherian> ::= **noetherian** <id> { **except** <id-list> }

<complete> ::= **complete** <id> { **except** <id-list> }

<acinfo> ::= **ac_info** <id> <id-list>

<kbweight> ::= **kb_weight** <weight-decl>

<weight-decl> ::= **vars =** <number> | <id> <id> **=** <number> { **,** <id> **=** <number> }*

<pr-ordering> ::= **pr_ordering** <id> <id> **>** <smaller-list> { **,** <id> **>** <smaller-list> }*

<smaller-list> ::= () | (<id-list>)

<display> ::= **display** <id> <funct-display> { **,** <funct-display> }*

<funct-display> ::= <id> **=** <display-format>

2.4.4 Task Syntax

Remember that tasks are just kept in the syntax to ensure the compatibility with the input specifications of the RAP system. They are not meaningful for the TIP system.

<task> ::= **task** <id>
{ <primitives> }
{ <unknown-decl> }
{ <goals> }
endoftask

<unknown-decl> ::= **unknown** <vars>

<goals> ::= **goals** { **exist** (<vars>) } <equation> { **,** <equation> }*

2.5 Context Conditions

2.5.1 General Remarks

For the sake of readability, context conditions are defined in an informal way. Note that an attributed grammar may be extracted from the text in a straightforward way. The position of a syntactic unit of interest within a context is indicated by schemes of specification text containing keywords, meta-variables for identifiers and dots (…).

2.5.2 Declarations of Identifiers

The occurrences of identifiers in specifications are divided into occurrences for *declaration* and for *use*. At a declaration occurrence, the *kind* of an identifier is associated with it (e.g. function-identifier, sort-identifier). The following rules cover the possibilities for declaring identifiers.

```
type T … endoftype
theorem T … endoftheorem
```

declares T as a type or theorem identifier respectively. Note that the declaration of T ends at the keywords **`endoftype`**, **`endoftheorem`**, respectively

```
sort … s …
```

declares s as a sort identifier.

```
func … f …: …
cons … f …: …
```

declares f as a function identifier.

```
all (… v …: …)
forall … v …: …
```

declares v as a variable identifier.

```
axioms … (A) …
```

declares A as an axiom identifier.

2.5.3 Use of Identifiers

The following rules cover the possibilities for using identifiers. Every identifier has to be declared (more precisely: its declaration must have been ended) before it is used. The kind of an identifier in its declaration and its use have to be the same. For instance if s has been declared as a sort identifier, it may not be used but as sort identifier.

```
func …: (s1, …, sn) s
cons …: (s1, …, sn) s
```

use s_1, …, s_n, s as sort identifiers.

```
hidden … h …
```

uses h as a sort or function identifier.

```
all (…: s)
forall …: s
```

use s as a sort identifier.

Identifiers occurring within a term are used as function or variable identifiers.

```
basedon … T …
display T …
noetherian T …
complete T …
ac_info T …
kb_weight T …
pr_ordering T …
```

use T as a type identifier.

```
noetherian … except … A …
```

uses A as an axiom identifier.

```
complete … except … f …
display … f = …
ac_info T … f …
kb_weight T … f = …
```
use f as a function identifier.

```
pr_ordering … f > (g1 ,…, gn) …
```
uses f, g_1, …, g_n as function identifiers.

2.5.4 Export and Import of Identifiers; Visibility

- An identifier is called *declared in* (type or theorem, respectively) T if its declaration takes place within the declaration of T (i.e. between **`type`** T and **`endoftype`** or **`theorem`** T and **`endoftheorem`**)
- Each sort or function i declared in T is called *exported* by T if it is not used in a **`hidden`** `… i` within T.
- A **`basedon`** `… T` statement *imports* all sort and function identifiers which are exported by T.
- Type and theorem identifiers are *global*, i.e. they are visible everywhere after their declaration.
- All other identifiers are *local*, i.e. they are visible between their introduction by a declaration or an import and the end of the unit where they were introduced.
- An identifier may be used only when it is visible.
- As soon as an identifier is visible, it must not be declared once more.

2.5.5 Miscellaneous Constraints

- Type-, theorem-, and axiom identifiers must be upper-case (as they are used in the command language).
- An identifier i in

  cons …: (…)i
  hidden … i …

  must have been declared (i.e., it may not have been imported) within the surrounding unit. Note that the restriction for “cons” means that all constructor functions of a sort have to be declared in the same type where the sort is declared.
- Each kind of <property> may use a type identifier at most once.
- In a display definition

  display … f = "s_1 _ s_2 … $s_{n\_}$ s_{n+1}" …

  the number of underscores has to match the number n of arguments in the declaration of f. Before, between and after the argument positions you can use n+1 (possibly empty) substrings. See section 2.10 for detailed information about display definitions.

2.5.6 Strong Typing

- If v is declared by

  all (… v …: s)
  forall … v …: s

  then the sort of the variable v is s.
- If f is declared by

  cons … f …: (s_1, …, s_n)s
  func … f …: (s_1, …, s_n)s

  then the sort of the term f(t_1,…,t_n) is s, and t_1, …, t_n have to be terms of the respective sorts s_1, …, s_n.
- A <number> has the standard sort “`Nat`” (see section 2.6).
- Within an equation $t = u$ or a rewrite rule $t \to u$, t and u have to be of the same sort.

2.6 Standard Types

The context conditions and the semantics of a given specification are evaluated after prefixing the specification with the declarations of some standard types. Important consequences are:

- The sorts and functions declared in the standard types are available in every user-defined type or theorem just by a “basedon”-statement.
- The system provides efficient standard implementations for the functions of the standard types. E.g. the arithmetical operations are executed on ground terms by machine instructions rather than by symbolic computation.

There are two standard types: `BOOL` (truth values) and `NAT` (natural numbers).

Their defining text is listed in appendix D.

2.7 Semantics of a System of Specifications

A type together with all the types referenced directly or indirectly by “basedon” forms a *type system*. The composed type of a type system is a large type defined by the union of the signatures and axioms of all the component types of the system. In our small example in section 1.3 the type system would consist of the types `SUM_NAT`, `NAT` and `BOOL`. The *denotational semantics*, or model semantics, of a type system is defined as the class of all term-generated, heterogeneous algebras where the rewrite-arrow (→) is interpreted as equality of objects and:

- with every sort a non-empty carrier set is associated
- every element of a carrier set can be represented by a term built of constructor functions or, in other words, we consider interpretations.
- with every function symbol a partial function on the corresponding carrier sets is associated
- every axiom holds for arbitrary instantiations of variable-identifiers with elements of the carrier sets.

For more details on the semantics see [Wi90].

The inductive proof algorithm is correct w.r.t. this semantics. So, a *proof* of a theorem guarantees that for every substitution of constructor-ground-terms for the forall-identifiers the instantiated theorem holds in the model class of the type system.

The differences between classical total algebra semantics and the TIP semantics are:

- The designation of constructor functions in the “cons”-part of a type are different. This constructor signature is used by the inductive theorem proving algorithm: it looks for case analyses which are complete w.r.t. the constructor functions.
- TIP considers the axioms to be oriented from left to right (as expressed by the arrow of the axiom-syntax). This does not induce differences in the model-theoretic semantics, but in the deduction calculus used. TIP relies on term-rewriting techniques. This means that validity and derivability coincide only completely when the axiom set forms a confluent (“Church-Rosser”) and noetherian term rewriting system. More details about this restriction can be found in [Hu85].
- TIP does not allow only hierarchical models as [Wi90] does. The language constructs for modular specifications (“basedon”, “hidden”) are used only for additional context checks, but not for the semantics. Nevertheless, TIP semantics is correct w.r.t. hierarchical semantics.

The operational semantics for algebraic specifications, in contrast, deal with conditional rewriting, unification and inductive theorem proving.

2.8 Semantics of a Theorem

The semantics assigned to a theorem is its validity in the inductive theory of every model of the model class of the composed type system (composition of all types referenced directly or indirectly by “basedon”), i.e. every constructor ground instance of the theorem is derivable from the axioms of the type system.

A given theorem is characterized by
- the type system it refers to
- a set of universally quantified variables
- a set of possibly conditional equations (goals).

A theorem is proved to be valid if it is possible to transform it to a set of empty goals by the following proof steps:
- Rewriting
- Complete case analysis (over one or several variables)
- Complete case analysis for an arbitrary term (which is added to the conditions)
- Application of an induction hypothesis
- Application of a lemma
- Application of a condition to the conclusion of its goal
- Deleting goals with a trivial conclusion
- Deleting goals with contradictory conditions

For a more detailed explanation of the proof steps and their use in the main proof algorithm see section 4.3 and section 4.6.

Our proof algorithm is sound but of course for non-trivial specifications it is not complete (due to Gödel's incompleteness theorem), i.e. every theorem which is proved by TIP is valid, but not every valid theorem of the inductive theory is provable by the TIP system.

2.9 User-Ensured Properties

A given type system may have some properties which can be used for several purposes, for instance to increase the efficiency of the proof algorithm. Besides that, they are interesting for a semantic analysis of specifications. Among them the following are of importance:

- the axiom set considered as a rewrite system is terminating (noetherian).
- the axiom set is sufficiently complete w.r.t the constructor functions.
- some of the function symbols are "AC" (associative and commutative).

The user is explicitly warned that an erroneous property clause may cause an unreliable proof!

Noetherian Property

A statement of the form

```
noetherian T except A1,…,An
```

has TIP assume that the rewrite system of T with the rules $A_1, \dots, A_n$ removed, is Noetherian. A rewrite system is called *Noetherian* if there are no infinite rewrite derivations $t_1 \underset{\mathfrak{R}}{\rightarrow} t_2 \underset{\mathfrak{R}}{\rightarrow} \dots$. This property is used for two purposes:

- to ensure that rewriting to normal form terminates, and
- to allow for '$\overset{+}{\underset{\mathfrak{R}}{\rightarrow}}$ (the transitive closure of the $\mathfrak{R}$-rewrite step relation) as an inductive ordering (see section 6.1).

If no `noetherian` clause concerning a type T is found in the input file, set TIP uses a built-in recursive path ordering with status to settle a Noetherian subsystem $\mathfrak{R}$ of T.

Complete Property

A statement of the form

```
complete T except f1,…,fn
```

assumes that every non-constructor function symbol declared in T but not in $\{f_1, \dots, f_n\}$ is *constructor* *complete*, i. e. they can be “evaluated” in every ground term by rewriting with the axioms. This information is used for generating complete case analyses.

If no `complete` clause concerning a type T is found in the input file, set TIP uses a built-in completeness check to determine all functions of T which are complete w.r.t. the constructor functions of their argument sorts.

Associative-Commutative Property

A statement of the form

```
ac_info T f1,…,fn
```

assumes that the function symbols $f_1, \ldots, f_n$ declared in T are associative and commutative. Of course, these function symbols must have two arguments and both argument sorts have to be the same as the result sort. Implicitly this statement adds two additional axioms for every function f_i:

$$f_i(f_i(x, y), z) \to f_i(x, f_i(y, z)) \qquad \text{(* } f_i \text{ is associative *)}$$
$$f_i(x, y) \to f_i(y, x) \qquad \text{(* } f_i \text{ is commutative *)}$$

But these axioms would form a non-terminating rewrite system. Thus, the AC information is used only for unification, matching and equality testing in TIP and we are proving theorems modulo these three equations.

Knuth-Bendix Properties

Statements like

```
kb_weight vars = n0  T1 f1 = n1, … ,fm = nm   T2 …
pr_ordering T f1>(…), f2>(…)
```

are used to define the weights and the ordering relation between function symbols needed for the Knuth-Bendix ordering. A `kb_weight` statement sets the weights of the variables (keyword “vars”) for all types and for functions of a specified type T. The n_i in the above statement are the weights (i.e. natural numbers).

A `pr_ordering` statement determines the precedence ordering for the function symbols of a type T. Inside the brackets you can specify a list of function symbols which are smaller than the function symbol left of the greater character. This information is only used by the Knuth-Bendix ordering [KB70]. If you do not want to use this special ordering you can safely ignore these kinds of property statements. For more detailed information about orderings available in TIP see section 6.1.

Please check carefully whether your specifications satisfy the stated properties. Incorrect information about completeness or AC may lead to wrong proofs! The termination property is less critical. This may only cause non-termination, unless you have chosen the rewrite ordering as termination ordering. In this case wrong termination properties may lead to wrong proofs. The other properties are safe.

2.10 Display Definitions

Display definitions are a mechanism to define a mixfix syntax for terms, i.e. a more readable representation for terms. For instance, it is convenient to read “if B then S1 else S2 fi” instead of “if(B,S1,S2)” or “3 + 4” instead of “add(3,4)”.

To avoid the difficulties of parsing such expressions, the current version of TIP uses mixfix notation only for system output, not for input. Moreover, the use of display definitions can be switched off and on (via the **SET PARAM DISPLAYUSED** command).

Display definitions are grouped together into units defining a representation for the functions of one specific type (`display T …`). A display definition for a function symbol f with n arguments has the form

```
f = "s1 _ … _ sn+1"
```

where the s_i are arbitrary (possibly empty) strings of printable ASCII characters. Obviously, double-quotes and underscore must not occur within such a string. Typical display definitions are:

`f = ""` - a suppressed constant (i.e. no arguments) `g = "_"` - a suppressed unary function symbol (typically an injection)

`h = "h(_)"` - a construction of the ‘normal’ function display

`if = "if _ then _ else _ fi"`
- the usual representation of an “if”-function

`add = "_ + _"` - the usual representation for addition.

For the standard types, also standard displays are defined. Display definitions can be inspected during a TIP session by a **SHOW DISPLAY <type name>** command.

For some special characters there is defined a special interpretation if they occur within a display-sub-string:

horizontal tab:	interpreted as a space in the display string
line feed:	ignored
backslash:	interpreted as a line feed during output.

The use of tabs for defining spaces in the display is necessary because spaces in the input specification are removed by the scanner.

For instance, the display definition

```
if = "if _\<tab><tab><tab>then _\<tab><tab><tab>else _\fi"
```

leads to a printout of the term “`if(B,S1,S2)`” as

```
if B
   then S1
   else S2
fi
```

The use of mixfix notation will only change the visible output of the TIP system. Its behavior during proofs will stay unchanged. As the display definitions do not take care of the precedence of function symbol, the correct bracketing may get lost.

3 TIP Command Language

3.1 General Structure of the System

The TIP system reads an input file containing text in the TIP specification language, performs syntax checks on this text, checks the noetherian and completeness property (if necessary), and subsequently enters an interactive interpreter for the command language. If the system detects syntax errors during the reading phase, the two subsequent phases are skipped.

The internal status of the system consists of a representation of the (type and theorem) input and a book-keeping of the values of the system parameters.

There are basically two operating modes and one editing mode of the system. In the STANDARD (top-level) mode there are commands available for:

- inspection of the system status;
- invoking the inductive prover for a theorem;
- starting the AC-Unify and AC-Match algorithm for an equation;
- complete a type or invoke the completion-based prover for a theorem;
- editing new or existing theorems;
- adjusting system parameters.

The section 3.4 explains the important commands which are available in STANDARD mode.

The so-called PROVE mode is entered only during the (inductive) proof of a theorem. As long as TIP is in this mode all the information about the current proof is present. Additional commands are allowed for:

- watching the internal steps of the inductive prover and inspection of the proof status;
- interactive control of the proof.

The PROVE mode is described more detailed in chapter 4. There the important commands of this mode are summarized, too.

When you edit an existing theorem or generate a new theorem then you are in the EDIT mode. (See command **THEOREMEDIT**[1] in chapter 5.) Here you can do the following additional things:

- entering function symbols and variables for building up a new equation (in a syntax-directed way);
- copy an existing equation;
- delete an equation;
- replace a subterm of an existing equation by a new term;
- construct or dismantle a conditional equation.

Section 3.5 contains a list of the standard TIP commands available together with a description of their effect. The commands for the EDIT mode are explained in chapter 5.

The interactive help facility (see section 3.3) can also be seen as some kind of mode of the TIP system. It is available from every mode and if you leave the help support, then you always return to the mode you came from. A graphical representation of all modes is given in figure 2. Besides, you can see in this figure which commands are used to switch from one mode to another.

3.2 TIP Command Syntax

The general syntax of TIP commands is defined by the following extended BNF grammar:

<TIP-command>	::=	<command> { <subcommand> }* { <option> }* { <parameter> }
<subcommand>	::=	<id>
<command>	::=	<id>
<option>	::=	/<id> { = <value> }
<parameter>	::=	<value> { . <value> }
<value>	::=	<id> | <number>

1. All TIP commands are printed upper case and bold in this paper.

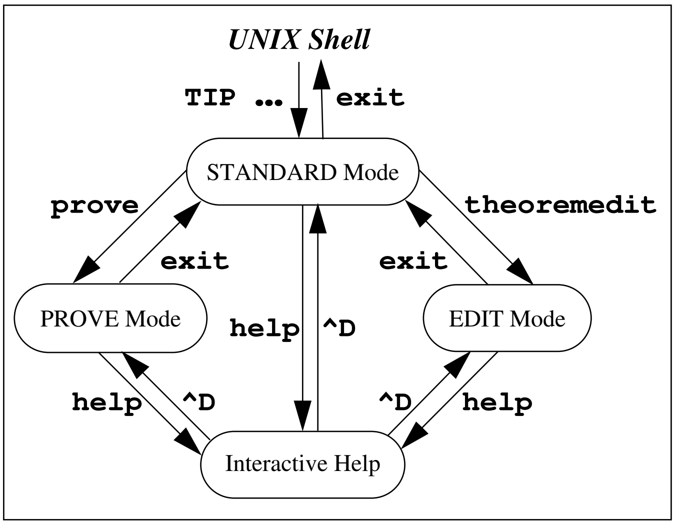


Fig. 2: The modes of the TIP system

The parts of a command are separated by blanks (space key). The '`/`'- and '`=`'-characters are also taken as separators[1]. <id> is defined as in the TIP specification language, but upper- and lower-case characters are not distinguished[2]. TIP commands, subcommands and options can be abbreviated. The abbreviation has to be unique for the current state.

3.3 Interactive Help About TIP Commands

The TIP system offers three features for on-line assistance about TIP commands: The **HELP** command and a menu-like technique for command browsing, the *tabulator key* ('`Tab`') for command and name completion and the *question mark* ('`?`') to see all possible commands for the current state.

The **HELP** command gives interactive access to TIP command documentation. It is designed for information purposes only, no action of the system is invoked. The manual text is organized in a treelike structure, the user is guided to select the interesting help text by entering keywords. For example if you type

```
TIP> help set
```

then you get the following information:

```
SET

    Set up system status.
    ---------------------
    The set command can be used in any mode for setting up system parameters
    (see SET PARAM). The only exception is the ORDERING parameter that has to
    be selected before entering the PROVE mode.
    This command serves additionally for changing the function attributes used
    in some of the induction orderings (see SET FUNCTION).

 FORMAT:

    SET subcommands

    WARNING: Command is not allowed in all modes.

Additional information available for:

FUNCTION      PARAM      CURRENT
TIP/HELP>
```

1. For file names (READ and WRITE command) the '`/`' character is not a separator but a normal part of the file name.
2. For file names (READ and WRITE command) upper- and lower-case characters are distinguished.

Normally an information text of the HELP command consists of:
- a short command description;
- a longer explanation including links to other commands;
- information about the syntax of this command;
- and a list of possible subtopics.

The empty input or Control-D returns the control to the calling mode. Otherwise the input is interpreted as a demand for information about a subtopic.

All these information texts are stored in the file named "tiphelp". If there are problems with the help facility maybe the installation at your computer system is not correct. See appendix B and appendix C for details about installation and necessary files and their locations.

The second assistance feature supports for completing incomplete commands, options or parameters by using the *space* and the *tabulator key*. If you enter the unique beginning of a command and then press the space key TIP will automatically complete this command. This also works for options and most of the arguments. The only exception here are arguments which are type or theorem names or numbers. The tabulator key does the same as the space key but it is more powerful. In addition to these features the tabulator key shows a list of all possible continuations if it is pressed two times. The tabulator key also works for completing (type or theorem) name arguments. So, when you enter an ambiguous beginning of a command the space key will lead to a warning while the tabulator key will cause a beep when pressed once and it will show all possible continuations if you press it the second time.

```
TIP> s
```

followed by the space key results in STANDARD mode in the following warning:

```
------WARNING: ABBREVIATION NOT UNIQUE
```

If you instead press the tabulator key twice you get the following behavior:

```
TIP> s
set   show
TIP> s
```

Now you can continue your command with an 'e' or a 'h' character. But you can also delete this 's' if this is not the beginning of the command which you had in mind.

Another example:

```
TIP> l
```

followed by the space or the tabulator key becomes in STANDARD mode (because it is a unique beginning):

```
TIP> list
```

If you now type the tabulator key twice you get the following result:

```
TIP> list
/file     ASSOC     GAUSS     REV       REVREV    SUM_NAT
/lafile   BOOL      HELP_RR2  REV2      RR2
/nofile   BREV      NAT       REVBREV   SEQ
TIP> list
```

These are the possible options of the list command (starting with '/') and all types and theorems which are currently loaded in the system. Now you can for example enter the (unique) beginning of a type or theorem name:

```
TIP> list bo
```

and if you now press the tabulator key you get the following result:

```
TIP> list BOOL
```

Finally you can ask for a list of all possible inputs for the current state by typing a *question mark* ('?') as command input. This is only allowed if the question mark appears in the place of a command, of an option or of a parameter. The tool gives a list of all possible continuations of the incomplete command or the current state. The user can select from this menu (or quit by entering an "end-of-file"-character[1]).

This feature is older than the command completion by the tabulator key. Thus, in many situations both

1. For UNIX this is Control-D.

help features behave quite similar. It depends on the user's preferences which one (s)he wants to use.

If you want to use this feature for getting an overview of all possible commands for the current mode just type a single question mark as command input. If you want to know the possible continuations or options of a command type "*<command> ?*" or "*<command> /?*" respectively. For example the input
TIP> *set ?*
results in STANDARD mode in the following output

```
*** Valid subcommands are:
*** Function, Param
TIP> SET
```

TIP lists all possible continuations of the **SET** command for the current mode. Those characters in this list which are written upper case are necessary for a unique abbreviation. In the above example it would be enough to type '*p*' for the subcommand **PARAM**. When you type the space key now the command will be completed. This kind of abbreviations makes it easier to type in long commands, e.g. the command **SET PARAM GENERALIZATION OFF** can be abbreviated by "`se p ge of`". Especially if you work more often with the system you will find these shortcuts very helpful.

Notice that TIP already repeats the correct part of the command input after the new prompt. This relieves the user from typing this part again.

The same facility is invoked if an incomplete command is entered, e.g. just typing "set" as input would lead to the same output as above, because **SET** is no complete command for the TIP system. If you do not remember a longer command exactly just type as much of the command as you know and see which continuations are possible.

3.4 Prompts, Control Keys and Command Editing

The system uses different prompts to indicate its current status

TIP>	Top-level prompt for standard commands.
TIP/PROVE>	Prompt during PROVE mode for standard and PROVE commands.
TIP/HELP>	Prompt during the interactive HELP facility for keywords leading to additional information for the current topic.
>>	Prompt during EDIT mode for text of a new equation. If the input starts with a dot '`.`' then the following text is interpreted as a command otherwise as part of the equation.
... `? (y/n/q)`	Prompt for a yes-no-quit decision during interactive proving.

The "empty input", i.e. pressing the RETURN key without entering other characters before, is in general interpreted by the system as something like "continue normally". It has the following effects:

- no effect	- in STANDARD mode
- equivalent to the PROVE command **STEP**	- in PROVE mode
- leave the HELP facility	- during HELP facility
- no effect	- in EDIT mode
- selection of "yes"	- for yes-no questions during interactive proving

The only control key interpreted by TIP is the "logical end-of-file" (Control-D in UNIX). It is taken as a request for terminating the current activity, depending on the current mode:

- leave the TIP system	- in STANDARD mode
- leave the PROVE mode[1]	- in PROVE mode
- leave the HELP facility	- during HELP facility
- stop completion of an incomplete command	- during command completion
- abort the theorem editing	- during EDIT mode

1. This means that all information about the current proof is lost.

The Control-C key can be used to kill the TIP system in case of problems or non-termination of proofs. This key is interpreted by your UNIX shell, not by TIP! After such an emergency stop there will be some links left in your current UNIX directory. You can either delete these links "by hand" or start the TIP system again. Further information about these links can be found in appendix C.

Since version 3.0 the TIP command parser includes the *GNU readline library*. This library allows you to edit the current input line and to go back to some former input line. Most of the Emacs[1] edit commands are available here. Please see any Emacs manual for a detailed description of these commands. To repeat some former command you can use '`!`' and the beginning of this command or '`!`' and the number of this command. This behavior is quite similar to some UNIX shells (like the "bash"). To see the list of the currently stored commands use the **SHOW HISTORY** command.

Entering "*`!pr`*" will repeat the last command starting 'pr' while "*`!17`*" will repeat command number 17 of the history list. At the end of every TIP session the last 60 commands of the user are stored in the file `.tiphistory` in the home directory of the user. These commands are available again in the next TIP session.

3.5 TIP Commands

The main part of every TIP command input is one command (sometimes called *master command*). This command can have one or several "subcommands", options and maybe one parameter. Options always start with the slash character ('`/`'). Two kinds of options are distinguished in TIP, the "select"-options and the "value"-options.

- *select-options* always are grouped in sets. One option of this set is the default. This option is used if no other option of this set is stated. An example are the **/TRACE** and **/NOTRACE** options of the **PROVE** command. Here **/NOTRACE** is the default. If you do not use the **/TRACE** option for a **PROVE** command then no trace (protocol) is generated. It is not possible to use both options at the same time. This would cause an error message.
- *value-options* are options which have an own argument. They are normally used to set some internal parameters. An example is the **/REWRITEMIN** option of the **SHOW PROVE** command. By writing **/REWRITEMIN = *n*** one can set the minimum number of rewrite steps for one line of proof output to *n* (see section 4.7.1 for details of this option). If you do not use this option for listing a proof then a default value is used for *n*. The same thing happens if you use the option but do not specify the *n*.

The parameter of a command is sometimes necessary and sometimes optional. And some commands do not have a parameter at all.

- Some commands have no parameter, e.g. the **DIR** command for listing all types and theorems;
- Some commands have to have a parameter, e.g. the **PROVE** command needs the name of the theorem which should be proved;
- Some commands may have a parameter, e.g. the **HELP** command may have another command as argument if one wants to get information about this special command at once.

Subcommands are used to specialize a command or a previous subcommand. Like parameters, subcommands can be optional sometimes. For example one can use **SHOW PARAM** to see all system parameters but one uses **SHOW PARAM ORDERING** to see only the ordering parameter. By way of contrast there are commands like the **SET** command which must have a subcommand to be complete.

For all command descriptions in the following chapters the possible subcommands, options and parameters are listed. Beyond, the default option for select-options is stated.

1. If you do not know or like the Emacs editor there is also a "vi" mode available. The default depends on the EDITOR environment variable of the user. You also can use the combination "ctrl-meta-j" for switching to the "vi" mode.

3.5.1 Standard TIP Commands

The following commands can be used in all modes of the TIP system. If there is a restriction for some special modes then this is stated in the explaining text. The line (in italic) below a command gives a very short description of this command.

HELP

Display information about TIP commands.

This command invokes an interactive manual about the TIP command language (see section 3.3). Empty input (RETURN key) returns from help mode. A **HELP** command without an argument lists the TIP commands and asks for entering a keyword (command name) to obtain further information. This will be recursively repeated for subcommands, options and parameters.

Parameter:	To obtain information about a specific command, this command (including subcommands and options) can be given as an argument to the **HELP** command.

DIR

Directory of known types and theorems.

A list of all known type and theorem names is displayed, i.e. a list of all identifiers for which a type or theorem declaration has been read from an input file and additional the two standard types `BOOL` and `NAT`.

Options:	**/THEOREMS** **/TYPES** **/ALL** (Default: **/ALL**)
/TYPES	Give a listing of the type names only.
/THEOREMS	Give a listing of the theorem names only.
/ALL	Give a listing of all type and theorem names.

LIST

Display the defining text of a type or theorem.

Given a known type or theorem name as argument, this command displays the textual representation of the type or theorem in a pretty-printed manner.

Options:	**/FILE** **/LAFILE** **/NOFILE** (Default: **/NOFILE**)
/FILE	Write the defining text to a file named <name>.rpp instead of the terminal.
/LAFILE	Write the defining text in LaTeX form to a file instead of the terminal.
/NOFILE	Write the defining text on the terminal.
Parameter:	The argument of a **LIST** command has to be a type or theorem name known in the current session. This means that a type or theorem declaration with this name is a standard type, contained in the file read initially by the system or read from a file by the **READ** command. The list of all known type and theorem names can be obtained by the **DIR** command.

PROVE

Start the inductive proving algorithm.

The **PROVE** command invokes the *Inductive Theorem Proving* procedure for verifying a theorem. The main proof algorithm can be influenced by the setting of some of the system parameters (see **SHOW PARAM** and section 4.5). After entering a **PROVE** command the system enters the PROVE mode. Here you can use a lot of special commands. But if you just want to do an automatic proof then use the **GO** command to start the proof and the **SHOW PROOF** command to see the proof listing if the system terminated successfully.

For detailed information about interactive proving see section 4.6.4. The proof algorithm is explained in section 4.6. See section 4.9 for more details on commands for the PROVE mode.

Parameter:	The argument of a **PROVE** command has to be a name of a known theorem. A list of all known type and theorem names can be obtained by the **DIR** command.
Options:	**/TRACE** **/NOTRACE** (Default: **/NOTRACE**)

/TRACE Produce a trace file. Each proof step is reported into a trace file whose name is **<theorem name>.trc**. The amount of information reported depends on the trace level which can be adjusted by the **SET PARAM TRACELEVEL** command.
The user interaction of the PROVE mode remains unchanged.

/NOTRACE Do not produce a trace file.

SHOW

Inspection of the system status and other parameter.

In any mode, the **SHOW** command can be used to have a look at the current values of the system parameters (by **SHOW PARAM**) and at the rules or type properties. During the work in the interactive PROVE mode, the **SHOW** command also serves for inspection of the current status of the proof (tree, goals, queues, hypotheses, etc.). For further information about interactive proving and system parameters see chapter 5.

Subcommands: **PARAM RULE DISPLAY PROPERTY FUNCTION HISTORY**

SHOW PARAM

Show system parameters.

This command displays the current values of all system parameters which can be used to influence the behavior of the inductive proof algorithm. Information about the value of one single parameter can be obtained by a **SHOW PARAM <parameter name>** command.

The system parameters can be set up by **SET PARAM <parameter name>** commands. For further information about the meaning of the parameters see section 4.5.

Subcommands: **LEMMA_USAGE HYPOTHESIS_USAGE REWRITE_USAGE GENERALIZATION ORDERING CA_GENERATION CA_USAGE HL_REDEX_SEARCH STRATEGY CUT_SELECTION GARBAGE_COL MIXFIX_USED INFO_LEVEL TRACE_LEVEL TERMINAL_KIND**

SHOW RULE

Show rewrite rules (axioms).

This command looks for the defining text of rewrite rules. It needs as an argument

- either a name of a type (displaying all axioms of this type)
- or a name of an axiom (displaying only one axiom).

Parameter: An argument for **SHOW RULE** may be:
- a type identifier known in the current TIP session,
- a pair consisting of a type identifier and an axiom name.
  Such pairs have to be written as **<type name>.<axiom name>**, i.e. separated by a dot and without any white spaces. For example `NAT.3` would be a correct argument.

SHOW DISPLAY

Show display definitions for mixfix output.

This command can be used for inspection of “display” statements issued in the input file. If there is no display defined for a function symbol, the standard (prefix) representation is shown. The display definitions are used by the system for pretty-printing of terms. The output in mixfix style can be switched on (or off) by a **SET PARAM MIXFIX_USED ON (OFF)** command. See also chapter 5 and section 2.10.

Parameter: The argument of a **SHOW DISPLAY** command has to be the name of a type known in the current TIP session.

SHOW PROPERTY

Show the known properties of a type.

This command displays “noetherian”, “complete” and “AC info” information about known types. Such properties have either to be defined within the input text or have been proved automatically by TIP. The second is done by two test criteria. The first criterion tests with a sufficient criterion if the axioms are terminating by the recursive path ordering. The second one tests if a function is specified completely w.r.t. the constructor functions. Both tests are explained in section 6.2.

Parameter: The argument of a **SHOW PROPERTY** command has to be a type identifier known in the current TIP session.

Options: **/FILE** **/NOFILE** (Default: **/NOFILE**)
/SHORT **/LONG** (Default: **/SHORT**)

/FILE Write the property information to a file instead of the terminal. The file name is the type name with the extension `.prp`.

/NOFILE Show the listing on the terminal only.

/SHORT Give only the results of the property checks (in short format).

/LONG Show the results of the property checks together with a more detailed diagnosis (e.g. containing uncovered cases for the completeness check).

SHOW FUNCTION

Show function attributes used for special orderings.

This command can be used to have a look at some special function attributes which are used for the lexicographic path ordering or the reduce term ordering respectively. Viewing and changing these attributes is only necessary for very tricky proofs.

Possible subcommands are **PERMUTATION** and **REDUCE** (see section 6.1 for further information).

SHOW HISTORY

Display the list of the last 60 commands used in TIP.

This command behaves quite the same way as the "history" command in UNIX. It displays a list of the last 60 commands that have been entered by the user. This list is useful for entering commands like "!*number*" or "!*string*" for repeating the command with this number or the last command starting with this string. (See also section 3.4.)

SET

Set up system status.

The **SET** command can be used in any mode for setting up system parameters (see **SET PARAM**). In PROVE mode, additional applications of SET become available.

Possible subcommands in the STANDARD mode are **PARAM** and **FUNCTION** (see section 6.1 for further information about the second subcommand).

SET PARAM

Set up system parameters.

By **SET PARAM <parameter name>** commands global parameters can be set up which change the behavior of the theorem prover. In the following, only the most important system parameters are explained. For a full explanation of system parameters see section 4.5.

Subcommands: **INTERACTIVE AUTOMATIC**
LEMMA_USAGE HYPOTHESIS_USAGE REWRITE_USAGE
GENERALIZATION ORDERING CA_GENERATION
CA_USAGE STRATEGY
CUT_SELECTION GARBAGE_COL MIXFIX_USED INFO_LEVEL
TRACE_LEVEL

SET PARAM INTERACTIVE

Adjust system parameters for interactive theorem proving.

This command adjusts seven system parameters at the same time. A "normal" environment for interactive theorem proving is set up. All non-trivial proof steps have to be confirmed by the user. This is very useful to keep the proof tree small. The effect of the **SET PARAM INTERACTIVE** command is the same as seven single **SET PARAM <parameter name> <parameter-value>** commands would have. Of course you can change the parameters step by step, too. For a full explanation of system parameters and their values for interactive proving see section 4.5 and section 4.6.4.

SET PARAM AUTOMATIC

Adjust system parameters for automatic theorem proving.

This command adjusts seven system parameters at the same time. The default environment for automatic theorem proving is set up. All possible proof steps are applied "in parallel". This causes a very fast growing proof tree. The advantage is that if a proof is found then always the shortest proof will be found[1]. The problem is that normally very long[2] and difficult[3] proofs cannot be found this way. Also, automatic theorem proving often leads to non-termination of the system. The effect of the **SET PARAM AUTOMATIC** command is the same as seven single **SET PARAM <parameter name> <parameter value>** commands would have. For a full explanation of system parameters and their values for automatic proving see section 4.5 and section 4.6.4.

SET PARAM INFO_LEVEL

Set up the amount of information to be displayed.

The **INFO_LEVEL** controls the amount of output displayed in PROVE mode. There are three possible levels: **BRIEF**, **STANDARD** (default), **DETAILED**. See section 4.5 for more details.

Parameter: One of the keywords **BRIEF**, **STANDARD**, **DETAILED**.

SET PARAM TRACE_LEVEL

Set up TRACE level.

The **TRACE_LEVEL** controls the amount of output which will be written to a trace file in PROVE mode, if the **/TRACE** option is used for a **PROVE** command. There are three possible levels: **BRIEF**, **STANDARD** (default), **DETAILED**, which are interpreted similar to the **INFO_LEVEL**. See section 4.5 for details.

Parameter: One of the keywords **BRIEF, STANDARD, DETAILED**.

SET PARAM DISPLAYUSED

Enable or disable the usage of mixfix notation.

Mixfix notation can be (de-)activated using this command. TIP displays terms in a pretty-printed manner according to the "display" statements in the input files. See also the **SHOW DISPLAY** command.

For defining a mixfix notation see also section 2.10.

Parameter: One of the keywords **ON, OFF**.

WRITE

Write or append types or theorems to UNIX files.

Write the text of a type or a theorem to a file of the UNIX file system. This is especially useful for new theorems which have been generated during the current TIP session with the help of the theorem editor. If no target file is specified then the listing is written to a file with the name of the type or the theorem along with the appendix '`.tip`'. For example **WRITE BOOL** will write the text of the type `BOOL` to a file named `BOOL.tip`. If you want to write it to an other file you have to use the **/TARGET = …** option. If this file already exists then the text is appended.

Parameter: The argument of a **WRITE** command has to be a type or theorem identifier known in

1. If the theorem is provable with the "default proof steps" of TIP and if there is enough memory available for the proof tree.
2. Because of the complexity, the necessary proof tree should not be to deep.
3. The system will not try any "tricky" application of proof steps (like new conditions) with these settings of the parameters.

the current TIP session.

Options: **/TARGET = <file_name>**

/TARGET Write the listing to the named file. If this file exists the listing will be appended. Otherwise this file will be generated.

READ

Read input files containing types and theorems.

The **READ** command is used to make types and theorems stored in a UNIX file available in the current session. The result is the same as if this file would have been added to the UNIX command line before starting the TIP system.[1] The file has to be a meaningful TIP input file which has not been read in the current session (otherwise you will get "multiple declarations" errors). If one wants to read a current input file again, maybe because it has been edited, then the READ command without an argument has to be used. In this case all types and theorems are undeclared and the input files are parsed again.

Parameter: The argument of a **READ** command has to be a UNIX file name of a file containing TIP specifications. If no argument is given then all input files are re-read.

ASSUME

Use a theorem as a valid lemma for all proofs of the following proof session.

The **ASSUME** command is used to make theorems visible as lemmas during all proofs of the current TIP session. This is equivalent to the use of **ENTER LEMMA /ASSUMED <theo_name>** (see section 4.9 for information about this command) at the begin of every proof. The options **/TRUE** and **/FALSE** are used to switch the general visibility of a lemma on or off. All currently assumed theorems are listed at the begin of every proof.

Parameter: The argument of an **ASSUME** command has to be a theorem identifier known in the current TIP session.

Options: **/TRUE** **/FALSE** (Default: **/TRUE**)

/TRUE Assume a theorem as a correct lemma for all following proofs.

/FALSE Reset the lemma status of a theorem.

EXIT

Leave the TIP system.

If the **EXIT** command is used in the STANDARD mode then it terminates the TIP system and returns control to UNIX. In PROOF and EDIT mode **EXIT** switches back to STANDARD mode. A "logical end-of-file" (Control-D) is equivalent to **EXIT**.

Other Commands in Standard Mode

For information about the commands **COMPLETE** (completion based proof algorithm), **UNIFY** (unify two terms using AC information) and **MATCH** (match a variable term with an unchanging term using AC information, too) see section 6.4 and section 6.3.

Moreover, the use of the **SET FUNCTION** and **SHOW FUNCTION** commands is explained in section 6.1. They are needed for setting and displaying function attributes which are used by different orderings.

3.5.2 Commands in the Start-up File

If the user wants to have some kind of standard settings for all his TIP sessions then (s)he can use a personal *initialization file* for the TIP system. This file must have the name '`.tiprc`' and it has to be in the home directory of the user.

This file may contain (correct) TIP commands which should be executed before the system enters the interactive STANDARD mode. Useful commands in this initialization file are:

1. Normally this command is "**tip** <file_name_1> … <file_name_n>" or "**TIP** <file_name_1> … <file_name_n>".

- read automatically all the files (= types and theorems) you are currently dealing with;
- change the system parameters to your favorite settings;
- assume the correctness of some basic lemmas;
- get an overview over all loaded types and tasks at the beginning of every TIP session (**DIR** command).

TIP prints the message "`Executing command file.`" when it starts to read the commands in the initialization file. You should not use wrong or incomplete commands in your initialization file because you only see the warnings and error messages but not the commands themselves during the TIP session. Of course unique abbreviations can be used. Besides, empty lines can be used in the `.tiprc` file because the empty input does not cause anything in the STANDARD mode. If you change to the EDIT or PROVE mode by a command the rest of the commands in `.tiprc` will be skipped.

Lower- and upper-case writing does only matter for file names.

3.6 Error Messages

The error messages issued by the TIP system contain a word indicating the severity of the error (like WARNING, ERROR, FATAL ERROR or SYSTEM ERROR) and a text which tries to fix the reason and the location of the error.

3.6.1 TIP Specification Language Errors

These error messages are composed of several lines. The line of the source text where the error was detected is displayed, together with its line number. In a second line below, the exact position is marked the error has been detected at. As a mark providing additional information a lexical element is used, which is in most cases equal to the source text above it, e.g.:

```
   120   func f : s
                  s
------ERROR: UNDECLARED IDENTIFIER.
```

In some cases the lexical element used for marking the error position is not equal to the source text above it. This means that the position where the error has been detected differs from the position where the error results from, e.g.:

```
   240      f(x,y)  = 3,
                  f
------ERROR: WRONG NUMBER OF ARGUMENTS.
```

Here the error was detected after reading the arguments of f, but the reason for the error is that f is used with a number of arguments different from its declaration. (The reader familiar with compiler construction may notice here that TIP uses a rigorous one-pass compiler architecture.)

Sometimes the system gives additional information about the error. If it is a syntactical error, the system prints a list of lexical elements allowed at the error position (after the word "`EXPECTED:`"), if a sort-conflict occurs, the conflicting sorts are named, etc.

```
     69     endoftype
            endoftype
------SYNTAX ERROR: ILLEGAL SYMBOL
      EXPECTED: , => & endoftheorem
```

Sometimes one error in the specification text may cause several subsequent errors. Especially, if the first error occurs in a declaration or in a "based on" statement. So, if TIP marks a lot of strange errors then start with checking the first error message, and continue in a top down manner.

3.6.2 TIP Command Language Errors

Error messages issued during the syntax check for commands have a simpler structure. In a blank line printed below the user's input the position of the error is marked by the sign '^'. (This line is sometimes omitted.) Then the line containing the error message follows, e.g.:

```
TIP> prove /file t2
             ^
------ERROR: UNKNOWN OPTION.
```

or

```
TIP> prove tesd
------ERROR: THEOREM NOT FOUND
```

4 Proving Theorems

This chapter is an introduction to the way TIP does the inductive theorem proofs. On one hand it is explained what the user can do to follow and control a proof. On the other hand it is explained what happens inside the TIP system, i.e. what does a single proof step result in and how is the proof algorithm build up. We think that this second kind of information makes it easier for the user to work with the system and to detect (and understand) its limitations.

To work with TIP it is necessary to understand the idea behind the constructor functions. For every sort there has to exist at least one distinguished function. These functions are called *constructors* of this sort[1]. And the concept is that every ground term has an equivalent constructor ground term. This is important for case analysis because then we only have to consider the constructors. Thus, if we have two constructors for a sort and we want to do a case analysis for a variable of this sort then we need only two cases and not one case for every function symbol with this sort as result sort. From a model-theoretic view this means that our proofs are restricted to constructor generated models.

In the first section of this chapter we give a general introduction to the idea and the principles of inductive proofs. Then the nodes of the proof tree, called "goals" are explained, because these are the objects the user has to deal with. The third section gives information about all the proof steps which are available in TIP. Afterwards the system parameters of the TIP system are explained which can be used to control the proof steps. Furthermore, the whole proof algorithm is documented. The next two sections are about the proof tree and the proof output. The first one shows the structure of a proof during the proving of a theorem, the second one is a pretty-printed listing of a successfully finished proof. Section eight is about the use of lemmas and section nine gives a summary of all commands which are only available in the PROVE mode. The last point is a list of useful hints for theorem proving with the TIP system. These hints are based on experiences from people who used the system for non-trivial proofs.

4.1 General Comments

The central idea of inductive proofs is to make a complete case analysis and to solve the cases either by applying the axioms or by additionally applying the induction hypothesis. This sounds easy and for some proofs this may work well. But for most of the examples this is not powerful enough. Sometimes one needs nested case analysis, generalization, application of lemmas or other rather special proof steps (see section 4.3). The TIP system allows to try different proof steps in parallel. Therefore all the information about connections between "new" equations and proof steps are kept in the proof tree (see section 4.4). Finally if a proof is successfully finished the user may like to see a pretty-printed listing of the proof. This is described in section 4.7.

For unconditional theorem proving one has only to deal with equations as lemmas or induction hypotheses. In the conditional case there may be some conditions which have to be fulfilled to have a correct application of the conclusion. This may cause problems if the check of satisfaction is non-trivial. For example one wants to apply the lemma:

$$t_1 = t_2 \;\Rightarrow\; l = r$$

The conclusion can be seen as a normal axiom of the form $l \to r$ which can be applied in our proof. But this application is only valid if the universal closure of $t_1 = t_2$ is valid. Therefore the equations in the conditions have to be verified, too. In TIP this is done in a lazy way. This means that evaluation is postponed to a later date. Every equation of the conditions forms a new "theorem"[2] and has to be proved by TIP. This again allows to use all possible proof steps of TIP (like induction) to verify the conditions.

Coding conditional theorems by a Boolean "implication" function[3] is much weaker because in this case

1. For example the sort **Nat** has the constructor functions `zero` and `succ(...)`.

2. More concrete it is a new goal (see section 4.2) which is combined with the original theorem by an "and"-connection in the proof tree (see section 4.4).

3. This was the normal way in earlier versions of the TIP system.

one needs the equivalent structure in the equation to apply such a theorem, e.g the above theorem would look like

$$\mathrm{impl}(\mathrm{equal}(t_1, t_2), \mathrm{equal}(l, r)) = \mathrm{true}$$

and it is not possible to apply it to a single occurrence of the term l. For the “really” conditional theorem above this would be possible.

Other papers describe conditional theorem proving as proving equations under a special context. There are some differences in comparison with our method of inductive theorem proving but the central ideas are quite similar. In both cases the conditions are a restriction of the terms for which the conclusion has to be valid. Also in both cases the conditions can be used to simplify the conclusion like “normal” rewrite rules. Of course this requires to deal with the problem of termination.

4.2 Goals

A “goal” consists of one or several (conditional) equations and some extra information like a number, a state and so on. This will be explained more detailed below. These goals form the nodes of our proof tree and they are the objects the proof steps work with. Normally a proof step takes one goal and generates one or several new goals. In the proof tree these new goals are appended as childs of the original goal.

The only exception of this principle is the simplification step (see section 4.3.1). This is done “inside” a goal. Using rewrite steps to transform an equation to its normal form is such a simple process that no new goals are generated for these steps. Instead the equations at the beginning and at the end of the simplification are kept together with a list of all the axioms which have been used for simplification.

The idea behind this handling of simplification is that humans prefer to deal with simplified forms of equations and that the simplification process itself is too trivial to be of interest for an inductive proof. For the final proof output (see section 4.7) the single rewrite steps can be recalculated from the information described above if the user is interested in a detailed proof listing.

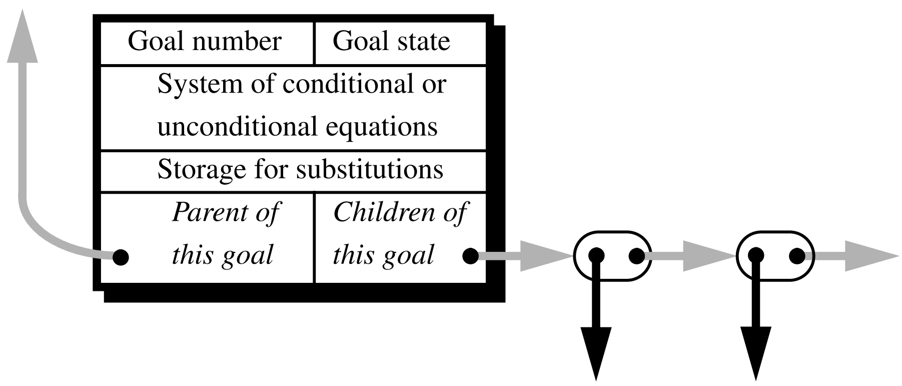


Fig. 3: The general structure of a goal.

The parent and children references are necessary to construct a proof tree (see section 4.4).

Beside these components there are some additional information stored for a goal, but they should not be interesting for the user (and they are also not visible for the user).

The state of a goal can be one of the following six possibilities:

- *Success* This state means that the goal is solved. Either because it was empty after removing all trivial equations during the simplification or because all children of one proof step applied to this goal have been solved.
- *Disproved* This state means that the goal is wrong. Either because a contradiction between two irreducible function symbols was detected during the simplifica-

	tion or because this information was inherited from the child of a proof step applied to this goal.
- *Failed*	This state means that the goal is neither solved nor disproved, but no proof step can be applied to this goal, e.g. we are stuck.
- *Stopped by user*	This state means that the user decided to cancel this goal, e.g. the **STOP GOAL** command was used.
- *Stopped by system*	This state means that the TIP system does not need this goal any more to prove the theorem. Either because an alternative branch of the proof tree was successful or because the goal was part of a branch which was skipped after some contradiction.
- *Active*	This state means that the goal is neither proved nor failed or stopped. This state again is divided into two subcases: - not simplified yet (*New goal*) - simplified and some possible proof steps are left (*Incomplete goal*)

Examples:

If a goal only contains the equation $\mathrm{f}(x, y) = \mathrm{f}(x, y)$, this trivial equation can be removed and the goal is solved. Its state will then be set to *Success*. A special procedure will check if this success can be inherited to the parent goal.

If a goal contains the equation $\mathrm{succ}(\mathrm{f}(x, y)) = \mathrm{zero}$ among others and if 'succ' and 'zero' are irreducible function symbols then the goal is proved to be wrong. Its state will then be set to *Disproved*. A special procedure will check if this disprove information can be inherited to the parent goal.

Every goal includes a substitution which is called *binding*. This substitution is a collection of all substitutions between this goal and the root of the proof tree. Such substitutions are only created by case analysis steps (see section 4.3.2). The binding substitution is necessary to calculate whether the application of an induction hypothesis is valid with regard to an underlying ordering. For more details see section 4.3.3.

In contrast to the types and theorems the goals use so-called *system-variables*. This is necessary because we need a (nearly) infinite stock of "new" variables for all the goals which are generated during a proof. Therefore the user-variables which are normally lowercase characters like 'x' or 'n1' are transformed to system-variables, e.g. '`X1`' or '`X17`'. The representation of a system-variable consists of an upper case X and a non-zero natural number. Only this form of variables appears in the PROVE mode.

There are two particular goals in TIP which should be pointed out. First there is the *start goal*. This goal is generated out of the equations of the theorem the user wants to prove. It normally has the goal number 1 and forms the root of the proof tree. If this goal is solved then the theorem itself is completely proved. The second special goal is called *current goal* and it is the goal which is currently worked on. Some commands use the current goal as default argument if no goal number is given. The commands **SHOW GOAL** and **SET CURRENT <goal -number>** can be used to inspect and set the current goal.

The current goal is important for the proof algorithm of the TIP system because it is build in a cyclic way. First one of the currently active goals is selected. This goal now is called the *current goal*. Now all possible proof steps are applied to this goal. The user's agreement is necessary if special system parameters are set to the "interactive" value. The new goals which are generated by the proof steps are added to the set of the active goals. After this the TIP system enters the command interpreter[1] and the user can inspect and influence the state of the proof. If the **STEP** command is entered then the next current goal is selected. This loop is repeated as long as the theorem is not proved or disproved or the set of active goals is not empty[2].

Beside the use of goals as nodes of the proof tree they are also stored in queues. Several goal queues are

1. If the **GO** command was used then TIP continues without entering the command interpreter.

2. The set of active goals may become empty if the current setting of the system parameter is too "weak" or if the user skipped some possible proof steps or stopped some goals. The third possibility is that the axioms in the specification are not complete w.r.t. the constructors.

used in TIP. The most important one is called "*wave*"[1] and contains all the goals which have not been processed. Newly generated goals are inserted into the wave queue. The place of this insertion depends on the search strategy (see section 4.5). You can use the **SHOW QUEUE** command to have a look at the wave queue. This queue always shows you which and how many goals there are left to be handled. The first goal in this queue will be the current goal for the next "round" of the proof algorithm.

Other queues are used to store stopped or solved goals. To see these queues you have to use the **SHOW QUEUE <queue name>** command. Possible <queue names> are:

- WAVE	those goals which are waiting for being processed
- SOLVED	those goals which have already been solved by TIP
- KEEP	those goals which have been processed but are neither solved nor disproved
- STOPPED	those goals which are not interesting any more because other goals have been solved or disproved and a branch of the proof tree was cut off

A goal is part of the proof tree as well as it is contained in one of the goal queues. The first one gives information about its place in the proof the second one depends mainly on the current state of the goal.

4.3 Proof Steps

Proof steps are the components from which the central proof algorithm of the TIP system is build up. A proof step takes one existing goal as "input" and generates one or several new goals as "output". These new goals are "sufficient" to the input goal, i.e. if the new goals can be solved then the input goal is solved, too.

The other way round this relationship is not necessarily correct. If the input goal is solvable this does not always mean that the goals generated by a proof step are also solvable. For example a generalization step (see section 4.3.5) may lead to a wrong goal but this does not mean that the original goal is wrong, too.

The application of proof steps can be influenced by setting the system parameters (see section 4.5). For example one can choose if the application of lemmas should be done always without asking the user or if one prefers an interactive way. In the following sections the system behavior for every proof step is presented for the interactive parameter settings because this is the point where the user has to know *what* (s)he has to decide. The interactive settings are those parameter values which are caused by a **SET PARAM INTERACTIVE** command (see also section 4.5).

By way of contrast the central proof algorithm (see section 4.6) cannot be changed by the user. Thus the order of the proof steps is fixed but their special behavior is not.

4.3.1 Simplification

The idea of simplifying a goal is to make it "easier". This should make it easier for the user to read the conditional equations. Another advantage of simplifying is that thereby it is possible to find out if the goal is solved or disproved. That's why simplification of a goal is one of the most important proof steps.

Unlike other proof steps the simplification of a goal does not generate new goals. It would be a waste of space to generate a new goal after every rewrite step. Therefore only a list of the steps which have been done during the simplification process is generated. Beyond the structure of the goal before and after the simplification is stored. All this information are necessary for the reconstruction of the full proof tree during the final proof output (see section 4.7).

The simplification of a goal consists of several different tasks:

- Deleting trivial equations[2]; if a conclusion is trivial then the whole conditional equation is removed
- Detecting contradictions in conditions and conclusion
- Rewriting the conditional equations to normal form[3]

1. It is called wave because it is the front line of interesting goals in the proof tree and it always moves "forward".
2. An equation is called trivial if both sides are syntactically equal or if they are equal under the AC relation (see section 6.3)
3. If the used specifications are not confluent then this normal form may be not unique.

- Expanding variables from conditions of the form `x = f(…)` if `x` does not occur in the term `f(…)`.
- Applying the conditions for simplifying the conclusion.
- Remove an irreducible function symbol on both sides if it is the outermost function symbol on both sides. This step is called *decomposition* and it may split one equation into several equations. See the example below.

An overview over of the simplification process is given in figure 4 and figure 4.

A *decomposition step* can be done for a goal of the form $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ if f is an irreducible function symbol, i.e. there exist no axioms for evaluating the function symbol f. But this means that the only way for solving this goal is that the arguments on both sides are equal. So, we can delete f on both sides. This is done by replacing the above equation by the following n new equations:

$$\begin{array}{c} s_1 = t_1 \\ \vdots \\ s_n = t_n \end{array}$$

An equation is *trivial* if both sides of the equation are equal modulo the AC relation (see section 6.3 for information on AC). If no AC properties have been specified in the input file (see section 2.9) then an equation has to have exactly the same term on both sides to be trivial.

A simple equation forms a *contradiction* if the outermost function symbols on both sides are different and both are irreducible. For the TIP system a function symbol is irreducible if there exist no axioms for this function symbol. So, there is no possibility to transform this equation to a trivial equation at any time. Normally only some of the constructor functions are irreducible. An example for a contradiction is the equation `succ(...) = zero` (based on the standard type `NAT`) because there are no axioms for the constructor functions `zero` and `succ`.

A goal is said to be *solved* if (and only if) all its equations have been deleted. The only way to remove conditional equations from a goal is a trivial conclusion or a contradiction in the conditions. In the first case we have a conditional equation with a valid conclusion. Therefore the conditions are no longer of interest and the whole equation can be removed. In the second case one of the conditions is “wrong” and it is a well-known fact that false implies everything.

A goal is said to be *disproved* if (and only if) there is a contradiction in a conclusion and this conclusion has no conditions. The second restriction is necessary because if the conclusion has some conditions then one of these conditions could become contradictory and then the goal would be solved!

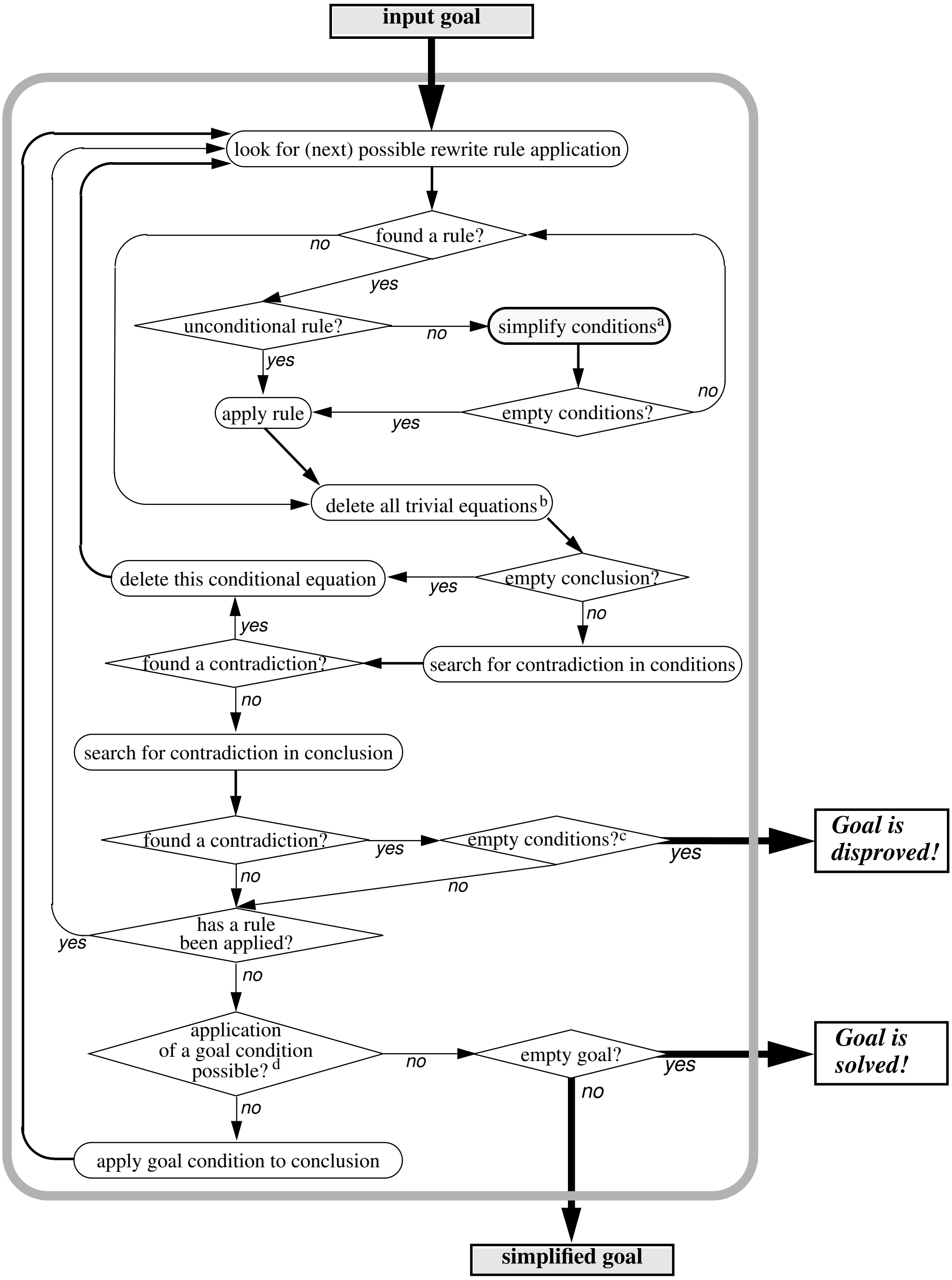
The information about solved or disproved goals is sometimes inherited upward to the father-goal in the proof tree. This depends on the sort of proof step which was used to generate the current goal. Also the status of the siblings of the current goal may be important. For example a solved case of a case analysis can only inherit the “solved”-information upward if all the other cases of this case analysis are solved too. Otherwise the heredity is delayed until the status of the other has changed. More detailed information about heredity can be found in the sections for the single proof steps.

What is not shown in detail in figure 4 are the expansion of variables and the application of conditions to the conclusion. Both steps are done at the end of the simplification process and if they are successful the whole simplification algorithm is called again because new rules may be applicable now.

The expansion of a variable means that all occurrences of a variable in a conditional equation are replaced by a term `t`. This is possible if a condition of the form `x = t` or `t = x` exists where `x` does not occur in `t`. Such conditions can be viewed as an assignment to the variable `x`. By the expansion the condition itself will be transformed to `t = t` and therefore it will be deleted as a trivial equation by the simplification process.

Application of a condition means that one of the conditions is used as a rule for rewriting the conclusion. Since the conditions are equations they could be applied left to right and right to left. To avoid non-termination we only use the “noetherian” direction, i.e. the direction in which terms get smaller according to the lexicographic path ordering[1] (see section 6.1). Another difference to ‘normal’ rewriting is that the variables in the conditions are the same as in the conclusion. Therefore we cannot apply an instance of

1. This ordering is also used to check the termination of axioms of the specifications.



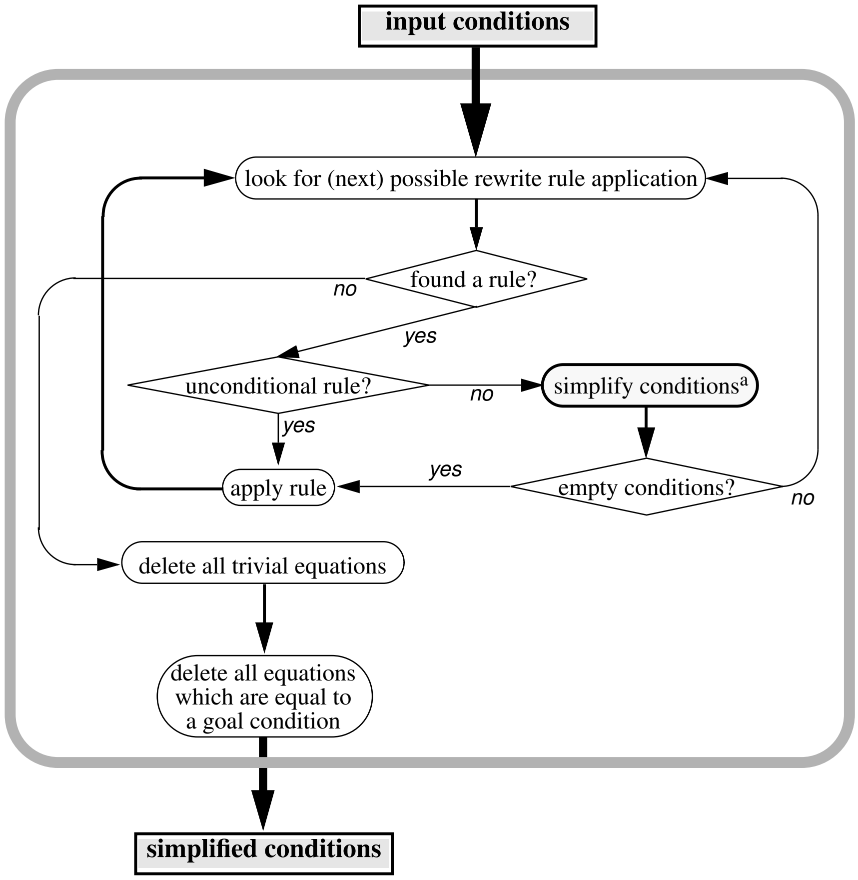
[a] A kind of *recursive call* to rewrite the conditions of a rule. The conditions of the goal are used here too. See figure 4.

[b] In the conclusions as well as in the conditions.

[c] Only conclusions with empty conditions can be disproved by a contradiction.

[d] Variable expansion or term replacement in the conclusion.

Fig. 4: The simplification of a goal.



[a] Recursive call of this procedure.

Fig. 5: The simplification of conditions.

a condition but the left hand side of our 'condition-rule' has to be syntactically equal to a term in the conclusion which should be simplified. To make this more clear we give an example: In the conditional equation

```
g(x) = true   =>   f(x,g(x)) = true
```

the condition could be applied because the term `g(x)` appears in the conclusion. So, the application would result in

```
g(x) = true   =>   f(x,true) = true
```

and maybe we could apply rewrite rules to `f(x,true)` now.
But if our conditional equation looks like

```
g(x) = true   =>   f(x,g(y)) = true
```

we cannot apply the condition because `x` and `y` are different variable and an instantiation of `x` by `y` would also concern the occurrence of `x` in the conclusion. Therefore this would not be a correct simplification step.

In some special proofs (see appendix E.5) the automatic rewriting of a term may not be desirable. For that reason the rewriting process during the simplification of a goal can be controlled by the system parameter **REWRITE_USAGE** (see also section 4.5). This parameter is also used to fix the handling of non-terminating rewrite rules. The following five parameter values are available:

Table 1. The possible values of the system parameter **REWRITE_USAGE**.

Parameter value	Explanation
NOETHERIAN	Only those rules are enabled which are known as terminating.
SEMIAUTOMATIC	Rules which are known as terminating are enabled for automatic rewriting; the others are presented to the user for a decision about their application.
INTERACTIVE	All rules are presented to the user for a decision about their application.
OFF	No rule is enabled for rewriting.

The distinction between terminating and non-terminating rules is important because otherwise the termination of the whole simplification process would be uncertain. The termination check (see section 6.2) for the rewrite rules is done by the TIP system after the reading of the input file. Information about the termination of some or all rules of a type can also be given by the user in the input file. So, if you really want to enable non-terminating rules for automatic rewriting you have to use the "noetherian" property in the input file on your own risk. See section 2.9 for information about user-ensured properties.

Usually goals are simplified immediately after their generation. The simplification process is documented to the user. The simplification of two goals (of a case analysis) is shown below as an example:

```
* TRY SIMPLIFICATION OF GOAL NO. 6.
*   GOAL REWRITTEN USING RULE NAT.18
*   GOAL REWRITTEN USING RULE BOOL.1
* CONTRADICTION IN PREMISES DETECTED => CLAUSE IS TRUE.
* THE IRREDUCIBLE SYMBOLS ARE >>> false <<< AND >>> true <<<
* GOAL NO. 6 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 7.
*   GOAL REWRITTEN USING RULE NAT.20
*   GOAL REWRITTEN USING RULE BOOL.2
*   TRIVIAL CONDITION REMOVED.
*   GOAL REWRITTEN USING RULE NAT.21
 GOAL IS SIMPLIFIED.
```

The first goal has been solved during the simplification process (by a contradiction in the premises of its conditional equation). The second one has been simplified without success or failure. The expression `NAT.18` means that axiom number 18 of the specification `NAT` was used for this rewrite step.

4.3.2 Complete Case Analysis

The complete case analysis is the most important step in induction proofs. It splits the proof of a goal in several new goals which should hopefully be easier to prove, because they are more special. A "side effect" of a case analysis is that the equations(s) of this goal can be used as induction hypothesis for all its child-goals. In the TIP system this new induction hypothesis is generated as a new rule for the set of induction hypotheses.

The principle of the term induction (for a single variable):

For an algebraic specification $T = (\Sigma, R)$, a predicate P over the terms of a sort $s \in S$ and a noetherian ordering $<_n$ over the terms the following is valid:

$$\forall x\colon R \mid\!- P(x)$$
$$\Leftrightarrow$$
$$\forall t\colon (\forall s, s <_n t\colon R \mid\!- P(s)) \;\Rightarrow\; R \mid\!- P(t)$$

Where $x, s, t \in W_\Sigma^s$

For the TIP system this principle is restricted to the constructor terms, because we are only interested in constructor term generated models. The predicate P has to be viewed as a (conditional) equation with one variable. On the other hand the above scheme can be extended to a set of terms instead of one single term t without problems.

In TIP a case analysis runs over one or several variables of the goal. These variable(s) are specialized by substituting constructor terms for them. All these constructor terms have to form a *complete* case analysis for the sort of the variable(s), i.e all possible ground terms have to be equivalent or more special (modulo R) than one of the constructor terms of the case analysis. The typical example for a case analysis are the natural numbers with the cases *zero* and *successor* of a new variable.

From a more practical point of view we distinguish between two different kinds of cases in a case analysis. First the *base cases*. They are build by substituting constructor terms which have no arguments. Examples would be `zero`, `succ(zero)`, `true` or `empty_list`. Such cases have to be solved without the application of an induction hypothesis. On the other hand there are the *induction cases* (or *induction steps*) which are build by substituting constructor function symbols with variable arguments. Examples are `succ(…)` or `append(…,…)`. Usually an induction hypothesis must be applied (see section 4.3.3) to solve such goals.

Case Analysis for Variables

The induction as it is known from mathematics does case analysis always for one variable by spliting the proof into a zero and a successor case. If we want to generalize this scheme then we can do case analysis for two or more variables simultaneously. Beyond, we can use more than two cases for one variable like the splitting into 0, 1 and everything greater than one. The only restriction we have to consider is that the case analysis still has to be complete. To give a short example the following case analysis for the two variable `x` and `y` (both natural numbers) is complete:

Case	Substitution for x and y	
1. Case	`x\zero`	`y\zero`
2. Case	`x\zero`	`y\succ(y')`
3. Case	`x\succ(zero)`	`y\y`
4. Case	`x\succ(succ(x'))`	`y\zero`
5. Case	`x\succ(succ(x'))`	`y\succ(y')`

Another important idea of TIP is to use the case analyses which are "hidden" in the axioms. The reason for this is that in most cases the functions are specified completely, i.e. the description of a function in the axioms forms a complete case analysis if one views only the argument-tuples of these axioms. There is an algorithm implemented in TIP which checks the completeness of function definitions w.r.t. the constructor functions (see section 6.2 for details about this test).

The TIP system generates such a case analysis by *unification* of a subterm of the goal with all axioms for the outermost function symbol of this subterm. If in the goal exists a non-constructor function with variable arguments then TIP would use each axiom of this function to generate one case of a complete case analysis for these variable(s). Of course this can only be done if TIP has the information that this function symbol is specified completely. Imagine we have a function `f` which is specified by the following three axioms:

```
f(zero) -> true,
f(succ(zero)) -> false,
f(succ(succ(n))) -> f(n)
```

In this case we (and the implemented completeness check) would say that `f` is specified *completely*. If a goal now contains an equation where `f` appears with a variable argument like

```
g(x) = not(f(x))
```

then the complete case analysis `[x\0]`, `[x\1]` and `[x\x'+2]` would be derived from the axioms of `f`. More technically described this kind of case analysis is done by unifying all axioms of `f` with a redex which has `f` as outermost function symbol. Every unification specializes the (conditional) equation, i.e. the variables of the redex have to be substituted. The main advantage of this method is that for every case it is guaranteed that at least one rule can be applied. Thus, a case analysis is chosen which brings us forward.

The usual way of doing case analyses is to select one variable, to determine its sort and to replace it in succession by all the constructor functions with this result sort. This corresponds to the "mathematical" way of doing case analyses in induction proofs. For example if you have a variable of sort `List` then TIP would create new goals for the two constructors `empty` and `append(…,…)`. If arguments are necessary for a constructor function then new variables of the matching sorts are generated. This forms a complete case analysis, because we are restricted to constructor term generated models and we use all existing constructor functions for the sort of the variable.

The TIP system can either use only one of these two kinds of generating case analyses or a mixture of both of them. This is selected by the system parameter **CA_GENERATION**. Its possible values can be found in table 2. If one uses the mixture of both methods then case analyses for variables are only done for those variables which are not involved in any case analysis generated by the rules of a specification. The reason is that we want to avoid multiple case analyses for one variable.

Table 2. The possible values of the system parameter **CA_GENERATION**.

Parameter value	Explanation
VAR_ONLY	Use only case analyses for single variables.
CONTEXT_ONLY	Use only case analyses which have been generated by the rules.
ALL	Use a mixture of both methods.

The following system output shows the behavior of this proof step in the interactive mode. It is influenced by the setting of the system parameter **CA_USAGE**. Its possible parameter are listed in table 3.

```
? POSSIBLE INDUCTION REDEX:
f(X1) = false => g(X1) = true

? Try induction for this redex? (y/n/q) y

* DOING A COMPLETE CASE ANALYSIS FOR GOAL NO. 1
* over variable X1 (generated by the rules of function f)
*   1. CASE: X1\zero [NEW GOAL: NO. 2]
*   2. CASE: X1\succ(zero) [NEW GOAL: NO. 3]
*   3. CASE: X1\succ(succ(X2)) [NEW GOAL: NO. 4]
```

First it is announced that a redex for a possible case analysis has been found. Then the equation(s) of the goal are printed with the variables of the redex marked in bold face. If the case analysis was generated by the rules of a function symbol then this function is printed underlined. If the user affirms the case analysis then TIP shows information about the original goal, the induction variables and the newly generated goals.

Table 3. The possible values of the system parameter **CA_USAGE**

Parameter value	Explanation
FULL	Use all redices for case analyses.
INTERACTIVE	Present all possible case analyses to the user and ask for decision.

In most cases the case analysis based on the rules is sufficient for proving theorems. Only if a variable appears separately on one side of an equation or if the surrounding function is specified incompletely then the case analysis over a single variable is necessary.

Case Analysis by Enriching the Conditions

The proof step which is described here is (up to now) only available by command and is not included in the central proof algorithm. The reason is that enriching the conditions is a non-terminating and a non-decidable process. Thus, the **ENTER CONDITION <goal-number>** command has to be used to apply this kind of case analysis.

Applying this proof step means to enrich the conditions with a new equation of the form `term = x` and then to do a case analysis for the new variable `x`. This is a valid proof step because `x` does not appear in the goal before and therefore it can be viewed as the definition of an abbreviation for a larger term and then distinguishing between the possible "values" of this term. For example if we want to distinguish between whether x>y holds or not then we add the equation `greater(x,y) = z` to the conditions and do a complete case analysis for the new variable `z`. This would lead us to the two cases where either `greater(x,y) = true` or `greater(x,y) = false` is added to the conditions. In the first case we can use the information that `x` is greater then `y` for proving the goal, while in the second case we must deal with the condition that `x` is not greater than `y`.

The TIP system handles the technique described above in the following way: After entering the command together with a goal number TIP asks for the term which should be used for the case analysis. This term is entered in the same (syntax directed) way as the equations are entered in the theorem-editor. (See section 5.1 for further information about this topic.) After the term is entered by the user the TIP system does the rest of the work automatically. First an equation of this term and a new variable is formed and added to the conditions of the stated goal. Then a case analysis is done for the new variable and all generated cases are simplified.

The only special feature when entering the term is that here we have to deal with system variables. So, we have to use the variable style of the TIP system. This means that the name of a variable always is an upper case 'X' followed by a (positive) number like 'X2' or 'X34'. Thus, if you want to use an existing variable of the goal for your term input then you also have to type the upper case 'X' and the number. Otherwise your variables would not be "shared" with the variables of the goal.

To make this process more clear we give a short example. Let us say we want to do a case analysis for the term `equal_Nat(X1,X2)` because this is the decision argument of our if-then-else function in the example:

```
TIP/PROOF> enter condition 1
GOAL NO. 1:
if(equal_Nat(X1,X2),X1,X1) = X1
 WITH (BINDINGS)
[]

> PLEASE ENTER THE TERM WHICH SHOULD BE USED FOR A NEW CASE ANALYSIS:
Insert (partial) equation. Use `?` for help or `.ABORT` to abort theorem
TERM: ?left side?
>> equal_Nat(X1,X2)
THIS TERM WILL BE USED FOR A NEW CASE ANALYSIS IN THE CONDITIONS:
   equal_Nat(X1,X2)
Do you want to do this? (y/n) y

* ADDING THE EQUATION
  equal_Nat(X1,X2) = X3 TO THE CONDITIONS OF GOAL 1
* DOING A COMPLETE CASE ANALYSIS FOR GOAL NO. 1
* over variable X3
*   1. CASE: X3\true [NEW GOAL: NO. 2]
*   2. CASE: X3\false [NEW GOAL: NO. 3]

* TRY SIMPLIFICATION OF GOAL NO. 2.
*   GOAL REWRITTEN USING RULE IF.1
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 2 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 3.
*   GOAL REWRITTEN USING RULE IF.2
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 3 SOLVED.
# THEOREM NO. 1 PROVED BY SYSTEM.

TIP/PROOF>
```

Without this case analysis in the condition it would not be possible to solve this example with TIP. The reason is that there are only two axioms for the function `if`. One with `true` as first argument and the other one with `false`. In the above example the rules are called `IF.1` and `IF.2`. These rules are:

```
{IF.1} if(true,x,y) -> x,
{IF.2} if(false,x,y) -> y
```

Thus, for the case that `x` and `y` are both non-zero we cannot apply one of these axioms because `equal_Nat` is defined recursively for two non-zero arguments. The only way to apply the axioms in a useful way is to do a case analysis for the first argument of the function `if`. Then we get for example goal 2 as:

```
equal_Nat(X1,X2) = true  =>  if(equal_Nat(X1,X2),X1,X1) = X1
```

Now the expression `equal_Nat(X1,X2)` in the conclusion is transformed to `true` by applying the condition. Thus the rule `IF.1` can be used to solve this goal. The same action is done in an analogous way for goal 3.

Up to now the start of case analysis by enriching the conditions is only possible for the user. An idea for the future is to find promising terms for case analysis by enriching the conditions by heuristics. This should be possible during the simplification of goals. If non-constructor terms prevent the application of rewrite rules - like `equal_Nat(X1,X2)` *in the example above - then these terms may be good candidates for such case analyses.*

4.3.3 Application of Induction Hypothesis

The generation and application of induction hypotheses is the main difference which distinguishes between inductive theorem provers and deductive theorem provers. For the first ones we need a storage of possible induction hypotheses and a noetherian ordering for checking the "smaller"-condition for the arguments.

With the TIP system we try to prove conditional equations, so each induction hypothesis is a conditional equation too. If we have a short look at the formula of induction

$$(\forall x\colon (\forall \boldsymbol{y}\boldsymbol{.}\boldsymbol{y} < \boldsymbol{x}\boldsymbol{:}\ \mathbf{P}(\boldsymbol{y})) \Rightarrow \mathrm{P}(x)) \quad \Rightarrow \quad \forall z\colon \mathrm{P}(z)$$

then we see that we are allowed to use the predicate P(...) for all arguments *y smaller than x*. If one uses a case analysis for the variable x with the cases *0* and *m+1* then in the *0* case do not have any instances of P as induction hypothesis, because there are no natural numbers smaller than 0. For the proof of the induction case P(*m+1*) we are allowed to use all instances P(n) with $n<m+1$. We call $[x \setminus n]$ the *application substitution* because we want to apply the instance P(n) of P(x). On the other hand we call $[x \setminus m+1]$ the *case analysis substitution* because it was used to generate the induction case P(*m+1*). It is necessary to consider this case analysis substitution, because we want to apply the induction hypothesis in a specialized case.

The idea for the TIP system is to compare the application substitution and the case analysis substitution. If the used ordering is monotone[1] then it is possible to compare these substitutions by applying both to the same term and then comparing the two substituted terms.

The only restriction for such a term is that all induction variables have to appear at least one time in the term. In TIP the left hand side[2] of the conclusion is used for this purpose, i. e. we use the application substitution and the case analysis substitution to generate two instances of the interesting induction hypothesis. Then the left hand sides of the conclusions of these instances have to be compared with regard to the used induction ordering. If the first one (from the application substitution) is smaller then the application of this hypothesis is valid.

Maybe this sounds a little bit difficult at the first moment, but for a general use of induction hypotheses one needs a possibility to check whether the application of an induction hypothesis is smaller than the "original" one w.r.t. the used ordering. And if one wants to compare a possible application with the "original" induction hypothesis - which is equal to the goal it was generated from - then information is needed about all substitutions for the induction variables in the meantime. The TIP systems stores these changes of the variables in a special structure called *binding*. Every goal has a binding which contains all substitutions[3] which occurred on the path from the start goal to this goal, i.e. you can view it as some kind of environment for the local substitutions. These bindings are displayed when you look at a goal

1. An ordering < is called monotone if it fulfills: $(t_1 < t_2) \Rightarrow (\mathrm{C}(t_1) < \mathrm{C}(t_2))$ for all "contexts" C.
2. If the hypothesis should be applied left to right; otherwise the right hand side would be used.
3. Substitutions occur only due to case analysis steps. Generalizations are stored in a different structure.

using the **SHOW GOAL <goal-number>** command:

```
TIP/PROOF> show goal 5
GOAL NO. 5 [STATE IS ACTIVE] :
X10 = add(X10,zero)
 WITH (BINDINGS)
[X1 = zero,
 X2 = succ(X10)]
TIP/PROOF>
```

After showing goal number, goal state and the (conditional) equation(s) this command displays the binding of the goal *in brackets*. In the example above the two variables of the start goal ‘X1’ and ‘X2’ have been substituted by zero or successor of a new variable respectively. This is very useful if the user wants to check if one of the hypotheses would be applicable for a goal. The **SHOW HYPOTHESIS <number>** command is used to see the available induction hypotheses:

```
TIP/PROOF> show hypothesis 1
Hypothesis 1:
   Generated from goal no. 1.
   add(X1,X2) = add(X2,X1)
TIP/PROOF>
```

The command output supplementary shows where this hypothesis was generated from. This is important because a hypothesis can only be applied to a goal if the hypothesis was generated from a goal which is a parent[1] of the current goal. Otherwise the hypothesis was generated in a sibling branch of the proof tree and therefore is not admissible for the current goal.

Another point is that the conclusion of an induction hypothesis is an equation. Therefore, we have two possible directions for applying it. In the TIP system it is handled by generating two directed induction hypotheses for every goal which is the “parent” of a case analysis. It also stores the information that these two “rules” are inverse to each other. This information is necessary to avoid cycles in the proof tree. The only restriction for the generation of such directed hypotheses is that the left hand side may not be a single variable. This difficult case is handled separately in the so-called twin steps (see section 4.3.6).

After this preparatory information we will now look at the algorithm for applying conditional induction hypotheses in TIP. First we need a place (*redex*) where a hypothesis could be applied. This search is done in a leftmost-innermost pass through the equations of the goal. For this first the conclusion of a goal and then the conditions are checked, because a useful application of a hypothesis is more probable for the conclusions. For every potential redex TIP tests if an induction hypothesis is applicable. This is done by the following conditions. Only if all of them are true the induction hypothesis is applied to the goal:

- was the hypothesis generated from a parent goal of the current goal?
- does the hypothesis syntactically match the potential redex? *(If this condition is true then it supplies the application substitution.)*
- use the application substitution and the binding of the current goal to generate two instances of the left hand side of the hypotheses. Is the first one smaller w.r.t. the used induction ordering?
- has the inverse direction of the actual induction hypothesis *not* been applied at an equivalent redex to any parent goal of the current goal? *(Otherwise we would just produce a loop.)*
- is the application of induction hypotheses restricted by the system parameter **HYPOTHESIS_USAGE** and is the current hypothesis concerned by this restriction? *(If the parameter value is INTERACTIVE then ask the user for a decision.)*

If all these conditions are true then the induction hypothesis is applied to the current goal. In the TIP system we use a kind of lazy evaluation of the conditions of induction hypotheses, i.e. every condition forms a new child goal. The application of a conditional induction hypothesis (with *n* conditions) will always generate *n*+1 new goals. Let us illustrate this with a schematic example:

1. All (recursive) parents of a goal are considered here. This is exactly the path of goals in the proof tree between the start goal and the viewed goal.

Imagine that we have a goal G: $P_1 \wedge \ldots \wedge P_n \Rightarrow t_1 = t_2$

together with its binding substitution β and that we have the conditional induction hypothesis H:

$$c_1 \wedge c_2 \wedge c_3 \Rightarrow l \to r$$

where the c_i are equations and l and r are terms. Suppose that the "parent" condition (first point is the list above) is fulfilled. Now we want to apply the hypothesis H to the goal G. Therefore the left hand side l of the hypothesis has to match with a subterm s of t_1. This subterm together with its surrounding context $C[\ldots]$ is equal to t_1. Let the application substitution be named σ, then we know:

$$l\sigma = s \text{ and therefore } C[l\sigma] = t_1$$

Now we have to check the ordering. Let $<_{\mathrm{TIP}}$ be the noetherian induction ordering used by the TIP system for this proof. (See table 5 for a list of possible orderings.) Then the following condition must be valid:

$$l\sigma <_{\mathrm{TIP}} l\beta$$

If this is true, i.e. the application instance is smaller than the original hypothesis, and if no other restriction avoids the application of the hypothesis then the following four "goals" will be created:

$$P_1 \wedge \ldots \wedge P_n \Rightarrow C[r\sigma] = t_2$$

$$P_1 \wedge \ldots \wedge P_n \Rightarrow c_1\sigma$$

$$P_1 \wedge \ldots \wedge P_n \Rightarrow c_2\sigma$$

$$P_1 \wedge \ldots \wedge P_n \Rightarrow c_3\sigma$$

The first one is the real application of the hypothesis as a rule, the three others are necessary to validate the conditions of the σ- instance of the induction hypothesis.

In the proof tree all these four goals are childs of our first goal and they form one *or*-branch where they are all connected with *and*, i.e. all four goals have to be solved so that the first goal is solved too. For more details on the proof tree of TIP see section 4.4.

The way induction hypotheses are applied or not is controlled by the system parameter **HYPOTHESIS_USAGE**. Its possible values are listed in table 4. The other important system parameter for hypotheses is **ORDERING**. This parameter determines which induction ordering is used to check whether an application of a hypothesis is smaller than the original goal. There are six different orderings available in TIP (see table 5). The generalized recursive path ordering is the default ordering of the TIP system. The other orderings may be useful for some special proofs.

Table 4. The possible values of the system parameter **HYPOTHESIS_USAGE**.

Parameter value	Explanation
ALL	Apply the induction hypotheses in both directions in an automatic way.
NOETHERIAN	Apply the induction hypotheses only in noetherian direction in an automatic way.
LR	Apply the induction hypotheses only from left to right in an automatic way.
INTERACTIVE	Present all possible applications of induction hypotheses to the user and ask for decision.
INTER_LR	Present only the left to right applications of induction hypotheses to the user.
OFF	Induction hypotheses are not applied at all.

Note: The system parameter **ORDERING** cannot be changed in the PROVE mode. Otherwise it may be possible to prove wrong theorems by switching between the orderings. All applications of induction hypotheses have to be checked with the same ordering. Therefore the user has to select the ordering before the **PROVE** command is used.

Table 5. The possible values of the system parameter **ORDERING**.

Parameter value	Explanation
LEXICOGRAPHIC	Use generalized recursive path ordering (= lexicographic path ordering).

Table 5. The possible values of the system parameter **ORDERING**.

Parameter value	Explanation
PARTTERM	Use subterm ordering.
MULTISET	Use multi-set ordering based on generalized recursive path ordering.
KNUTH-BENDIX	Use Knuth-Bendix ordering.
REDUCETERM	Use an experimental semantic ordering.
REWRITE_STEP	Use the rewrite ordering $\xrightarrow[R]{+}$ defined by the axioms R. This ordering is only admissible if the axioms of R are terminating.

If the parameter **HYPOTHESIS_USAGE** for the application of induction hypotheses is set to the value **INTERACTIVE** then TIP asks the user in the following way:

```
* CURRENT GOAL: NO. 4
even(X2) = false => odd(X2) = true

? POSSIBLE RULE USAGE: Hypothesis 1:
 even(X1) = false => odd(X1) = true
? AT
even(X2) = false => odd(X2) = true


? Apply this rule? (y/n/q) y
* REWRITE STEP USING INDUCTION HYPOTHESIS 1
*** 1. premise transformed to goal. [NEW GOAL: No. 5]
even(X2) = false => even(X2) = false

*** conclusion transformed to goal. [NEW GOAL: No. 6]
even(X2) = false => true = true
```

First TIP shows the possible induction hypothesis and then the current goal with the redex marked boldface. This means that the conclusion of the hypothesis could be applied from left to right[1] as rewrite rule at the marked redex of the goal. If the user agrees then this proof step is performed. The condition of the hypothesis (under the application substitution `[X1\X2]`) forms the conclusion of the first new goal, the goal itself after the rule application forms the second new goal.

4.3.4 Application of Lemmas

In TIP the name "lemma" is used for theorems whose validity is already known. Maybe this terminology is different to other papers. One can view lemmas as auxiliary theorems. The application of lemmas is quite similar to application of induction hypotheses. A lemma is a (conditional) equation, therefore we have again to check the conclusion for an application in both directions. In difference to a hypothesis a lemma does not have to fulfil any "smaller"-condition.

Similar to hypotheses a lemma is stored as two conditional directed rules in TIP. The application of these rules is controlled by the system parameter **LEMMA_USAGE**. Its possible values are listed in table 6. The search of possible redices for lemma applications is done leftmost-innermost.

Table 6. The possible values of the system parameter **LEMMA_USAGE**.

Parameter value	Explanation
ALL	Apply the lemmas in both directions in an automatic way.
NOETHERIAN	Apply the lemmas only in noetherian direction in an automatic way.
LR	Apply the lemmas only from left to right in an automatic way.

1. Otherwise TIP would print: `POSSIBLE RULE USAGE: Hypothesis 1 (inv):` for an application from right to left.

Table 6. The possible values of the system parameter **LEMMA_USAGE**.

Parameter value	Explanation
INTERACTIVE	Present all possible applications of lemmas to the user and ask for decision.
INTER_LR	Present only the left to right applications of lemmas to the user.
OFF	Lemmas are not applied at all.

There are two possible ways of making lemmas available in a proof. The first one is to use the **ASSUME <theorem name>** command before starting a proof. Then the user takes the responsibility for the correctness of this theorem. The theorem then will be used as a lemma in all proofs of this proof session.

```
TIP> assume help1

Correctness of theorem HELP1 is assumed by the user!

TIP> prove t1
ASSUMED THEOREM (from inputfile): HELP1
# CURRENT THEOREM: NO. 1
```

One can see that the user always is informed when a theorem is used as a lemma in the new proof. The **ASSUME** command cannot be used during a proof. If one wants to add new lemmas during a proof then one has to use the second possibility for making lemmas available. In this case one uses the **ENTER THEOREM** command together with a known theorem name as parameter. If no argument is used then the theorem-editor is called for entering a new lemma. Up to now it is only possible to enter unconditional lemmas by this way. For details about syntax-directed input of the theorem-editor see section 5.1. If one uses the **/ASSUME** option for this command then again the user takes the responsibility for the correctness of this theorem. Otherwise TIP system delays the current proof and tries to prove this new theorem first. After a successful proof of such a theorem the TIP system returns to the "original" proof and the auxiliary theorem is now available as a lemma.

Let us assume that a user realizes the need of the commutativity of the multiplication while proving some theorem. Of course (s)he knows that this is a valid lemma:

```
TIP/PROOF> enter lemma /assumed
>PLEASE ENTER EQUATION OF NEW THEOREM:
Insert (partial) equation. Use '?' for help or '.ABORT' to abort theorem
THEOREM: ?left side? = *
>> mult x y mult y x
###> 'x' was declared as variable of sort Nat
###> 'y' was declared as variable of sort Nat
EQUATION OF NEW LEMMA:
mult(x,y) = mult(y,x)

TIP/PROOF> step
* CURRENT GOAL: NO. 1
add(mult(X2,X1),mult(X1,X3)) = mult(add(X2,X3),X1)

? POSSIBLE RULE USAGE: LEMMA 2:
 mult(X4,X5) = mult(X5,X4)
? AT
add(mult(X2,X1),mult(X1,X3)) = mult(add(X2,X3),X1)


? Apply this rule? (y/n/q) y
* USING LEMMA 2
*** conclusion transformed to goal. [NEW GOAL: No. 4]
add(mult(X1,X2),mult(X1,X3)) = mult(add(X2,X3),X1)
```

First the new lemma is entered (in a syntax directed way) and then it is immediately applied to the current goal. This way of introducing new lemmas is very useful for lemmas for which their necessity is not known from the beginning of the proof. Thus, when the need of a special lemma becomes obvious during a proof session it is possible to "generate" it without loosing the current state of the proof. In this case one can either postpone the proof of the lemma and generate a "proved" version or one interrupts the current proof to verify the new lemma first. It is possible to do such kinds of interrupts more often, i.e.

it is possible to recursively enter new lemmas for the proof of a lemma and so on.

4.3.5 Generalization

Generalization denotes that a term of the goal equation(s) is replaced by a new variable. If this term occurs several times in the equation(s) then every occurrence is replaced by the variable. We can say that a generalization is the opposite of a substitution because instead of substituting a variable by a term we replace a term by a new variable.

Maybe at first sight it looks a little bit strange to have a proof step which makes a goal more general. It does not seem to be a good idea to prove something more difficult. The reason for this step is that if our goal is very special then our induction hypothesis is special too. Then it may happen that we cannot find any fitting redex for this hypothesis. A "simpler" induction hypothesis would be useful for this case. But if we want to have a more general hypothesis then we have to prove a more general goal as well.

There are proofs where it is necessary to prove something more general than the original "problem" just to have a more powerful induction hypothesis available. Of course if the generalized goal is proved then the original "special case" is proved too. On the other hand if a generalized goal is disproved then TIP cannot conclude that the "special case" is disproved, too. More likely the generalization has been too general. Then this branch of the proof tree is skipped and the other branches are continued.

We want to illustrate this with the example that was already used in section 1.3. There we tried to prove the formula of Gauß and after some proof steps we got the following goal (the addition should be bracketed left to right in the following expressions):

$$\boldsymbol{x} + \frac{(xx) + x}{2} = \frac{(xx) + x + \boldsymbol{x} + \boldsymbol{x}}{2}$$

If we do a case analysis for this goal for the variable x then the goal itself becomes an induction hypothesis and the simplified child goal for the "successor" case looks like:

$$y + \frac{(yy) + y + y + y}{2} = \frac{(yy) + y + y + y + y + y}{2} \qquad \text{with x=y+1}$$

Here the structure of the goal and the hypothesis are too complicated. Thus, it is not possible to apply the hypothesis to the goal. If the term $(xx) + x$ is generalized to a new variable z then this situation changes. The generalized version of the goal has the following form:

$$x + \frac{z}{2} = \frac{z + x + x}{2}$$

If we again do a case analysis for the variable x the (simplified) "successor" case looks like:

$$y + \frac{z}{2} = \frac{z + y + y}{2} \qquad \text{with x=y+1}$$

Here it is easy to see that the hypothesis can be applied to the generalized goal. And if we can prove this generalized goal then we have also proved the original goal, because it is just an instance. Therefore generalization is a valid and a useful proof step for inductive proofs.

A problem with automatic generalization as a proof step of TIP is that there exists a large number of possibly generalizations. Most of these more general goals are likely "wrong". Therefore we implemented five different values for the system parameter **GENERALIZATION** (see table 7). With this parameter the user can control what kind of terms should be generalized and if this should be done in an interactive or automatic way.

Table 7. The possible values of the system parameter **GENERALIZATION**.

Parameter value	Explanation
ALL	Do all possible generalizations in an automatic way.
MULTIPLEONLY	Generalize in an automatic way only such terms which appear several times in the goal.

Table 7. The possible values of the system parameter **GENERALIZATION**.

Parameter value	Explanation
INTERMULT	Generalize in an interactive way only such terms which appear several times in the goal.
INTERACTIVE	Do all possible generalizations in an interactive way.
OFF	Do not generalize anything.

Terms which are *never* generalized are single variables and constructor terms because it is never useful to replace a variable by another variable and in most of the cases it is no good idea to remove the information which is contained in a constructor term. So, if you want to generalize a constructor term you have to enter this as a new lemma.

4.3.6 Inflation in Connection With Generalization (Twin Step)

The 'twin step' is a very special proof step for applying induction hypotheses and theorems of the form $x \to t$ where x is a variable which occurs in t. The problem with such kind of rules is that there is a single variable on the left hand side and that it is non-terminating. Moreover, this rule could be applied to every (sub-)term of suitable sort of a goal again and again. This would lead to an extreme large number of (mainly useless) child goals.

In the normal application of theorems and hypotheses this kind of rules are not considered. The application of such inflating rules is only useful if afterwards some "new" progress in the proof is possible. For the TIP system this progress has to be a generalization. Only if the inflation step enables a generalization step for several terms which was not possible before then this combined proof step is applied.

Or the other way round: If you would like to generalize several subterms of a goal and one of the "interesting" redices has not the right form then a *twin step* is applied if it is possible to get this necessary redex form by inflation with a hypothesis or a theorem of the form $x \to t$.

For example if we have an induction hypothesis (or lemma) $x \to \mathrm{f}(\mathrm{g}(x))$ and a goal $\mathrm{g}(x) = \mathrm{h}(\mathrm{g}(x), x)$ then the inflation of the third x would allow the generalization of the term $\mathrm{g}(x)$ for all occurrences of x. This may be necessary if all the axioms for the function h have a constructor term as first argument and we want to use them for a case analysis. The inflation step would lead to $\mathrm{g}(x) = \mathrm{h}(\mathrm{g}(x), \mathrm{f}(\mathrm{g}(x)))$ and then it is generalized to $z = \mathrm{h}(z, \mathrm{f}(z))$. Now a case analysis for the new variable z is possible.

A more formal description of this inflation criterion would look like this:

Let C, c, D and E be contexts and x and y are variables. Assume we have an "inflation rule" $y \to \mathrm{C}[\mathrm{c}[y]]$ and a goal G of the form:

$$\mathrm{D}[\boldsymbol{x}] = \mathrm{E}[\mathrm{c}[x]]$$

We would like to generalize $\mathrm{c}[x]$ but the occurrence of x on the left hand side does not have this context c[…]. This recommends to apply the rule to x on the left hand side. Our goal G now has the form:

$$\mathrm{D}[\mathrm{C}[\mathbf{c}[\boldsymbol{x}]]] = \mathrm{E}[\mathbf{c}[\boldsymbol{x}]]$$

Now it is possible to generalize $\mathrm{c}[x]$ to the new variable z for all occurrences of x. So, at the end of a twin step the goal would be:

$$\mathrm{D}[\mathrm{C}[z]] = \mathrm{E}[z]$$

The main effect of this proof step is that it "moves" contexts, i.e. the context c[…] on the right hand side of the equation is removed and replaced by a C[…] on the left hand side. This may be useful if a "case analysis for the context E[…]" is necessary.

Now we want to use the double reversing of a sequence as a practical example for the twin step. In this proof, the induction hypothesis $x = \mathrm{rev}(\mathrm{rev}(x))$ should be applied to the goal

$$\mathrm{rev}(\mathrm{conc}(\mathrm{rev}(\mathrm{X3}), \mathrm{append}(\mathrm{X2}, \mathrm{empty}))) = \mathrm{append}(\mathrm{X2}, \mathbf{X3})$$

only at the last occurrence of X3. This inflation results in the provisional goal

$$\mathrm{rev}(\mathrm{conc}(\mathbf{rev}(\mathbf{X3}), \mathrm{append}(\mathrm{X2}, \mathrm{empty}))) = \mathrm{append}(\mathrm{X2}, \mathrm{rev}(\mathbf{rev}(\mathbf{X3})))$$

and here the term $\mathrm{rev}(\mathrm{X3})$ is generalized to the new variable $\mathrm{X4}$:

rev(conc(**X4**, append(X2, empty))) = append(X2, rev(**X4**))

Now a case analysis for X4 is successful because all axioms for the function symbol `conc` need a constructor term as first argument.

4.3.7 Conditional Rewrite Steps

If the conditions of a rewrite rule can be evaluated to `true` then this rule is applied during the simplification process like any unconditional rewrite rule. Otherwise, if the conditions are not evaluated to `false`, i.e. the conditions of the rewrite rule cannot be evaluated completely, then this conditional rewrite rule is applied in a "lazy" way like the lemmas or the induction hypotheses. Every condition of the rewrite rule forms a new goal. These "condition"-goals are siblings of the goal which is created by the application of the rewrite step. This special handling of conditional rewrite steps is very useful if a lemma or an extra induction proof is necessary to solve these conditions.

For example it would be possible to apply the conditional rewrite rule

$$\text{equal\_Nat}(a, b) = \text{ true} \Rightarrow \text{f}(a, b) \rightarrow \text{true}$$

to the goal

$$\text{equal\_Nat}(y, x) = \text{ true} \quad \Rightarrow \quad \text{or}(\mathbf{f}(\boldsymbol{x}, \boldsymbol{y}), \text{g}(x, y)) = \text{ true} \ .$$

The condition of the necessary instance of the rewrite rule would be '$\text{equal\_Nat}(x, y) = \text{ true}$'. But this is no direct conclusion of the goal condition '$\text{equal\_Nat}(y, x) = \text{ true}$'. Therefore an extra goal is generated for validating the condition. The TIP protocol for this proof step would look like this:

```
? POSSIBLE APPLICATION OF A CONDITIONAL REWRITE RULE: Rule DEMO.1:
 equal_Nat(X2,X1) = true => f(X2,X1) -> true
? AT
equal_Nat(X1,X2) = true => or(f(X2,X1),g(X2,X1)) = true

? Apply this conditional rewrite rule? (y/n/q) y

* USING THE CONDITIONAL REWRITE RULE DEMO.1
 equal_Nat(X1,X2) = true => f(X1,X2) -> true

*** 1. premesis transformed to goal. [NEW GOAL: No. 2]
equal_Nat(X1,X2) = true => equal_Nat(X2,X1) = true

*** conclusion transformed to goal. [NEW GOAL: No. 3]
equal_Nat(X1,X2) = true => or(true,g(X2,X1)) = true
```

The first new goal (for the condition of the rewrite rule)

$$\text{equal\_Nat}(\text{X1}, \text{X2}) = \text{ true} \quad \Rightarrow \quad \text{equal\_Nat}(\text{X2}, \text{X1}) = \text{ true}$$

can now be solved by a simple induction proof and the second new goal is solved by simplification. Thus, the above example goal is solved too.

4.4 Proof Tree

The proof tree is a kind of memory for the total state of a proof. It contains information about all the applications of proof steps, about the state and the relationship of goals and about the solving or failing of single goals or whole branches of the tree. The proof tree has a special structure which is called "*and-or-tree*" (see figure 6).

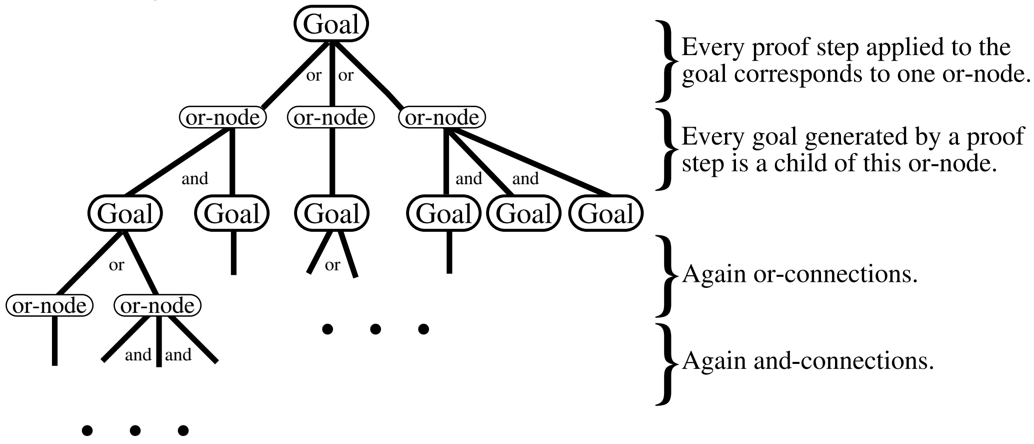


Fig. 6: The and-or-structure of the proof tree.

This is because the branches are alternating and-connections and or-connections. A goal can have one or several child-nodes which each result from a proof step and which form an or-connection because they describe alternative ways of solving this goal. These "*or-nodes*" again have one or several child-goals which form an and-connection because one single proof step (i.e. one or-node) can generate several new goals. An obvious example therefore is the case analysis. One case analysis leads to several new goals which of course are connected by "and". On the other hand several different case analyses form alternative ways of solving the goal. This is represented by the or-nodes.

This "and-or" information is used to determine when a goal is *solved*:

A goal is solved

- if it contains no more equations (= solved by simplification).
- if all "and-nodes" of one of its "or-nodes" are solved (= solved by upward inheritance).

If one views the nodes of the proof tree as Boolean values and the inner nodes as Boolean expressions formed according to their "or-nodes" and "and-nodes"[1] then the whole proof tree can be seen as a large boolean expression. Solved goals should correspond to the Boolean value *true*, all other nodes shall be Boolean variables. Then we can say that a theorem is solved if the Boolean expression corresponding to the proof tree of the theorem can be evaluated to *true*.

For disproving a theorem the situation is a little bit more difficult because some proof steps need not to inherit the information about a contradiction to their parent-goal. "Strict" proof steps like case analyses or unconditional lemma or hypothesis applications inherit the disprove information to the parent goal. But for example a generalization step or a conditional proof step may lead to a "wrong" goal without disproving the parent goal.

You can have a look at the proof tree with the **SHOW TREE** command. Often the tree grows too large to be useful for the user. In those cases it may be very helpful to use the **SHOW TREE <goal-number>** command to see only that part of the proof tree which is "below"[2] this goal. Besides, to make the proof tree more clear solved branches are skipped. Only one solved goal is displayed instead. If you want to see the whole tree anyway then you have to use the /FULL option when listing the proof tree.

The proof tree is displayed in ASCII format by indenting the child branches. The current goal is marked by "`<–CURRENT`" in this listing.

1. Inner nodes correspond to a disjunctive normal forms, e.g.: $G = ((g_i \wedge g_j) \vee g_k \vee (g_l \wedge g_m \wedge g_n))$
2. The goal itself and recursively all the childs of this goal.

```
TIP/PROOF> show tree

Proof Tree:

 GOAL 1 ACTIVE
   using COMPLETE CASE ANALYSIS
    1. Case: X1\zero
     GOAL 2 SUCCESS
    2. Case: X1\succ(X2)
     GOAL 3 ACTIVE
       using IND. HYPOTHESIS No. 1
         GOAL 4 ACTIVE  <- CURRENT
           using GENERALIZATION
             GOAL 7 ACTIVE
           using GENERALIZATION
             GOAL 8 ACTIVE
           using COMPLETE CASE ANALYSIS
            1. Case: X2\zero
             GOAL 9 SUCCESS
            2. Case: X2\succ(X29)
             GOAL 10 ACTIVE
       using COMPLETE CASE ANALYSIS
        1. Case: X2\zero
         GOAL 5 SUCCESS
        2. Case: X2\succ(X9)
         GOAL 6 ACTIVE

TIP/PROOF>
```

With the TIP system it is possible to prove several theorems in parallel. For example if you notice the need of an auxiliary lemma while proving your theorem then you can enter and prove this lemma and then continue proving your original theorem with the help of this new lemma. This is possible because a theorem inside TIP is completely described by its proof tree and all the goal queues. For further information on this topic see section 4.8.

4.5 System Parameters

There are several global parameters of the system which can be inspected by the **SHOW PARAM** command and changed by a **SET PARAM <param-name> <param-value>** command. The behavior of the TIP system depends very much on the setting of these system parameters. They control the application of proof steps, the form and amount of output presented to the user and other system specific properties.

First we give an overview of all TIP system parameters, afterwards every parameter is explained together with all its possible values.

SYSTEM PARAMETER	Explanation of the parameter
• LEMMA_USAGE	Control the application of lemmas.
• HYPOTHESIS_USAGE	Control the application of induction hypotheses.
• REWRITE_USAGE	Control the application of rewrite rules (= axioms).
• GENERALIZATION	Control the generalization of subterms.
• STRATEGY	Control the search strategy for the current proof.
• CUT_SELECTION	Control the pruning of the proof tree.
• CA_GENERATION	Control the generation of case analyses.
• CA_USAGE	Control the use of redices for generation of case analyses.
• CYCLE_LIMIT	Detect and prevent the occurrence of cycles in the proof tree.
• ORDERING	Fix the ordering which is used for induction proofs.
• GARBAGE_COL	Switch garbage collection on or off.
• MIXFIX_USED	Switch between function display in prefix and mixfix format.
• INFO_LEVEL	Select the amount of information which is presented to the user.

The two commands **SET PARAM AUTOMATIC** and **SET PARAM INTERACTIVE** are abbreviations for setting nine important system parameters simultaneously to suitable values for automatic or interactive proving respectively. Table 8 shows the default values for automatic and interactive parameter settings. If this “stack” of parameter settings is not exactly what you need then, of course, it is possible to change single parameter values with the **SET PARAM <param-name> <param-value>** command one by one.

The parameter values which are printed boldface in table 8 are the default values when starting the TIP system. Mainly these are equivalent with the automatic proof mode. A more detailed explanation of all system parameters and their parameters follows on page 52.

Table 8. The system parameters and their default values.

System Parameter	Default Value for SET PARAM AUTOMATIC	Default Value for SET PARAM INTERACTIVE	Other Possible Values for This System Parameter
LEMMA_USAGE	**ALL**	INTERACTIVE	NOETHERIAN, LR, INTER_LR, OFF
HYPOTHESIS_USAGE	**ALL**	INTERACTIVE	NOETHERIAN, LR, INTER_LR, OFF
REWRITE_USAGE	**NOETHERIAN**	SEMIAUTOMATIC	INTERACTIVE, OFF
GENERALIZATION	**MULTIPLEONLY**	INTERMULT	ALL, OFF, INTERACTIVE
STRATEGY	**BF**	DF	---
CUT_SELECTION	**OFF**	SINGLESTEP	REDUCED_CA
CA_GENERATION	**ALL**	ALL	VAR_ONLY, CONTEXT_ONLY
CA_USAGE	**ALL**	INTERACTIVE	---
CYCLE_LIMIT	**2**	NO_CYCLES	*any natural number*
ORDERING	---	---	**LEXICOGRAPHIC**, PARTTERM, MULTISET, REDUCETERM, KNUTH-BENDIX, REWRITE_STEP
GARBAGE_COL	---	---	ON, **OFF**
MIXFIX_USED	---	---	ON, **OFF**
INFO_LEVEL	---	---	BRIEF, **STANDARD**, DETAILED

LEMMA_USAGE

This parameter allows you to control how and which lemmas are applied (see section 4.3.4).

The six possible parameter values are:

- **OFF** Lemmas are not applied at all.
- **ALL** Every possible application is done automatically. This may lead to a large number of child goals if there are many lemmas.
- **NOETHERIAN** Only the noetherian direction of every lemma is used for automatic lemma application.
- **LR** Only the left to right direction of every lemma is used for automatic lemma application. Thus the equation is treated like directed rule.
- **INTERACTIVE** Every possible application of a lemma is presented to the user and (s)he has to decide if this application is meaningful.
- **INTER_LR** Only the left to right application of a lemma is presented to the user and (s)he has to decide if this application is meaningful

HYPOTHESIS_USAGE

This parameter allows to control how and which induction hypotheses are applied (see section 4.3.3).

The six possible parameter values are:

- **OFF** Induction hypotheses are not applied at all. *(This is not very useful!)*
- **ALL** Every possible application is done automatically. This may lead to a large number of child goals if there are many induction hypotheses.
- **NOETHERIAN** Only the noetherian direction of every induction hypothesis is used for automatic hypothesis application.
- **LR** Only the left to right direction of every induction hypothesis is used for automatic lemma application. Thus the equation is treated like directed rule.
- **INTERACTIVE** Every possible application of an induction hypothesis is presented to the user and (s)he has to decide if this application is meaningful.
- **INTER_LR** Only the left to right application of an induction hypothesis is presented to the user and (s)he has to decide if this application is meaningful

REWRITE_USAGE

This parameter allows you to control how and which rewrite rules are applied during the simplification step (see section 4.3.1). For every rewrite rule the TIP system has information if this rule may lead to non-termination[1] or not. See section section 6.2 for further information about the termination test.

The five possible parameter values are:

- **OFF** Rewrite rules are not applied at all. *(This is not very useful!)*
- **ALL** Rewriting is done fully automatic. *(This may lead to non-termination if there are rules which are not noetherian.)*
- **NOETHERIAN** Only the noetherian rewrite rules are used for automatic rewriting. The other rules are ignored.
- **SEMIAUTOMATIC** Only the noetherian rewrite rules are used for automatic rewriting. The other rules are presented to the user for a decision about application.
- **INTERACTIVE** Every possible application of a rewrite rule is presented to the user and (s)he has to decide if this application is meaningful.

GENERALIZATION

This parameter allows you to control how and which terms are generalized (see section 4.3.5). Generalization can also be caused by the ***GENERALIZE*** *command (see section 4.9).*

The five possible parameter values are:

- **OFF** Generalization is not used at all.
- **ALL** Generalization is done fully automatic. *(This leads to an extremely large number of new goals which mainly are useless!)*
- **MULTIPLEONLY** Only non-constructor subterms which appear at least two times in a goal are generalized automatically.
- **INTERMULT** Only non-constructor subterms which appear at least two times in a goal are presented to the user for a decision if this generalization would be useful.
- **INTERACTIVE** Every possible generalization is presented to the user and (s)he has to decide if this generalization is useful.

STRATEGY

This parameter selects how the proof tree is searched (see section 4.4).

1. The information “non-noetherian” does not mean that this rule always causes endless loops. It just states that the TIP system or the user cannot guarantee that this rule will never cause endless loops. So, the noetherian information is sufficient but not necessary to avoid loops.

The two possible parameter values are:

- **BF** Breadth-first search. This is organized by putting new goals at the end of the queue of active goals. Thus the tree is worked off plane by plane. *Useful for automatic theorem proving because no branch can be missed.*
- **DF** Depth-first search. This is organized by putting new goals at the front end of the queue of active goals. Thus, the tree is worked off branch by branch. *Useful for interactive theorem proving because it is more human like to finish one case of a case analysis before treating the next one.*

CUT_SELECTION

This parameter selects if some branches of the proof tree can be skipped (see also section 4.6.3).

The three possible parameter values are:

- **OFF** The full proof tree is build up. No branches are skipped.
- **SINGLESTEP** For every goal only one single proof step is done. All other possible proof steps are skipped for this moment. They can be applied after a user-caused backtracking. *Useful only for interactive theorem proving because then the user can select the most promising proof step for continuation and all other steps are skipped. This keeps the proof tree very small.*
- **REDUCED_CA** Equivalent case analysis branches are skipped. Such equivalent branches normally appear when several case analyses are done in varying order in parallel branches. *Useful for automatic theorem proving because a lot of unnecessary case analysis are skipped. This is uncritical because every case analysis which is skipped already exists somewhere else in the proof tree.*

CA_GENERATION

This parameter selects how the case analyses are generated. See section 4.3.2 for more detailed information about using the axioms for building up context-generated case analyses.

The three possible parameter values are:

- **ALL** Both methods are used for generating possible case analyses. For every variable of the goal not appearing in any context-generated case analysis a "simple" constructor based case analysis is done.
- **CONTEXT_ONLY** Only case analyses which have be generated by unification with the axioms are used.
- **VAR_ONLY** Only case analyses for single variables are used. The different cases are derived from the constructor functions of the sort of the variable.

CYCLE_LIMIT

This parameter selects how often a (growing) pattern may be repeated in consecutive goals, before this branch of the proof tree is skipped (for a more formal description see section 4.6.3, point 3).

The possible parameter values are:

- **NO_CYCLES** The test for cycles is suppressed.
- **<natural number>** After this number of systematically growing goals the branch of the proof tree is cut. (A useful default value is 2.)

CA_USAGE

This parameter allows you to control which case analyses are used.

The two possible parameter values are:

- **ALL** Every possible case analysis is generated and used automatically. This normally lead to a large number of child goals, but it is necessary for the completeness of the search strategy and for detecting the 'shortest' proof.

- **INTERACTIVE** Every possible case analysis is generated and presented to the user. Then (s)he has to decide if this (set of) variable(s) is a good choice for the next case analysis.

ORDERING

This parameter selects which ordering is used to test if the application of an induction hypothesis is smaller then the original hypothesis. There are six orderings available in the TIP system.

The best experiences we made with the lexicographic path ordering. Only for very "tricky" proofs the other orderings may have special advantages. See also section 6.1 for more details about the orderings. The six possible parameter values are:

- **LEXICOGRAPHIC** Use the lexicographic path ordering.
- **PARTTERM** Use the "simple" subterm ordering.
- **MULTISET** Use the multi-set ordering based on the lexicographic path ordering.
- **KNUTH-BENDIX** Use the Knuth-Bendix ordering.
  Hint: A lot of values have to be specified in the input file for a meaningful use of this ordering. See section 2.9 for specification details.
- **REWRITE_STEP** Use the rewrite ordering which is based on the axioms of the input specifications. The use of this ordering is only possible if the axioms are terminating.
- **REDUCETERM** Use an experimental version of the part-term ordering.
  Warning: Up to now the correctness of this ordering has not been proved. So, proofs which are done with this ordering should be treated carefully.

GARBAGE_COL

This parameter allows you to switch garbage collection on or off. There is a simple garbage collector (using the PASCAL dispose) built into TIP. It works simply by "disposing" unneeded parts of the heap as soon as possible. Switching on this garbage collector slows down the system, but is recommended for very large proofs.

The two possible parameter values are:

- **ON** Use garbage collection.
- **OFF** Do not use garbage collection.

MIXFIX_USED

This parameter allows you to change the display format of terms. It controls whether terms are displayed using the standard functional (prefix) notation or whether a mixfix output according to user-defined "display"-definitions is produced. Of course this is only possible if there exists a display definition for the function symbol in the input file. The "display"-definitions can be inspected by a ***SHOW DISPLAY <typename>*** *command. See also section 2.10.*

The two possible parameter values are:

- **ON** Use mixfix notation.
- **OFF** Use prefix notation.

INFO_LEVEL

This parameter allows you to control the amount of information shown in PROVE mode. This means especially how much information is given about proof steps and goals.

This parameter controls the amount of information shown in PROVE mode on a trace file, too.

The three possible parameter values are:

- **BRIEF** Only very few information is shown to the user.
- **STANDARD** The important proof information is shown to the user.

- **DETAILED** A lot of details are shown additionally to the standard output.

4.6 Proof Algorithm

4.6.1 General

In section 4.4 we introduced the single proof steps. Now we want to explain how these steps are combined to the proof algorithm of the TIP system and what else is done in this algorithm. The proof algorithm of TIP is designed to support automatic proving as well as interactive search for the proof.

There are a lot of different tasks which have to be handled while proving a theorem:
- the management of the goals and the selection of the next current goal;
- apply all or some possible proof steps to the current goal;,
- determine which effects the solving or disproving of a goal has for their parent-goals;
- give the user information about the progress of the proof;
- deal with the control input of the user if necessary (for interactive features).

Of course the second point is the most difficult one in this list. But also reporting the current state and the progress is an important task for interactive theorem proving. On the other hand it is difficult to describe this in a formal way. Thus, for getting an impression of the system output during an interactive proof we recommend to have a look at the examples in appendix E. Besides, the control input is not explained in this chapter. See section 4.9 for a description of all possible user commands.

The other three tasks will be explained in this chapter. First we will give a general overview for these tasks. In the following section a much more implementation-oriented description is presented.

First we want to present a very simplified description of the proof algorithm:

Generate the **start goal** from the theorem and put it into the set of active goals;
<u>While</u> the *start goal* of a proof is not solved
<u>do begin</u>
Select a goal ***G*** from the set of active goals;

<u>If</u> ***G*** is not simplified <u>then</u> **simplify *G*** by rewriting, applying the conditions to the conclusion, variable expansion and decomposition steps;
<u>If</u> ***G*** contains only trivial equations now (i.e. all equations have syntactically identical sides)
<u>then</u> goal ***G*** is **solved**
<u>else</u> make a complete **case analysis**[1] (several new child goals)
or apply any existing **unconditional induction hypothesis**[2] (only one new child goal)
or apply any existing **conditional induction hypothesis**[2]. (several new child goals)
or apply any existing **unconditional lemma** (only one new child goal)
or apply any existing **conditional lemma** (several new child goals)
or **generalize** any subterm of the goal ***G***; (only one new child goal)
or apply any non-decidable **conditional rewrite rule** (several new child goals)
or use a combination of **hypothesis application and generalization**; (only one new child goal)

Simplify all new child goals;
<u>If</u> a child goal contains only trivial equations <u>then</u> it is **solved**
<u>else</u> it is inserted into the set of active goals;
<u>If</u> all child goals of one proof step are solved <u>then</u> goal ***G*** is **solved** too;
<u>If</u> ***G*** is solved <u>then</u> check recursively if the parent goals of ***G*** are also **solved** now;
<u>end</u>

In contrast to this systematic description the test if a goal is solved or disproved is part of the simplification procedure because in most cases the check for an empty goal or a contradiction is only useful for simplified goals. Another difference is that new goals normally are simplified immediately after their generation to see if they are already solved. And the last simplified description in the algorithm above is the point that the implemented algorithm allows several different applications of every proof step.

1. A case analysis includes the generation of a new induction hypothesis.
2. If the application is a smaller instance of the hypothesis w.r.t. the used induction ordering.

4.6.2 Implementation

This section will give a much more implementation oriented description of the proof algorithm of the TIP system. You should be familiar with the notations of goals (see section 4.2) and the proof tree (see section 4.4).

Goal Management

All the goals of one proof are managed in three goal-queues:

- A queue called "*wave*" containing all goals which are active (= waiting for being processed).
- A queue called "*solved*" containing all goals which have already been solved.
- A queue called "*keep*" containing all other goals. These are the stopped and disproved goals.

The important queue for the proof algorithm is the "wave" because it contains all the active goals and always the first goal of this queue is selected to be the next "current goal" which is processed in the proof algorithm. The **SHOW QUEUE** command will show you the wave-queue. To see the other queues use the **SHOW QUEUE [WAVE | KEEP | SOLVED | STOPPED]** commands. The output will look like:

```
TIP/PROOF> show queue
Queue of active goals (queue "wave") :
4, 5, 7, 8, 9, 10, 11, 12
```

You can resort the wave-queue or activate stopped goals with the **SET CURRENT <goal-number>** command. This command moves the specified goal to the top position of the wave queue. Therefore this goal will be the next one to be processed. If the goal moved from the keep-queue to the wave-queue with this command then you do a kind of manual backtracking because the TIP system will never do such a transfer by itself.

If you try to move a goal to the top of the wave-queue which is already completed, i.e. either solved or all possible proof steps have already been applied, then you will get an error message:

```
TIP/PROOF> set current 6
------ERROR: GOAL IS ALREADY COMPLETELY DONE
TIP/PROOF>
```

These queues are directly connected with their theorem. If you switch to the proof of another theorem (maybe for proving an auxiliary lemma) then you will also have different goal-queues.

Effects of Solving or Disproving Goals

If a goal is solved or disproved then this may also lead to a solving or disproving of its parent-goal. This works recursively and if the "start goal" of a theorem is solved then the theorem has been proved. Exactly the other way round, if the "start goal" is disproved the theorem was proved to be wrong.

These two procedures for evaluating the consequences of solving and disproving goals are important. Both of them are part of the simplification procedure because only simplified goals are tested for being solved or disproved.

Procedure GoalSuccess:

The parent-goal of a solved goal is solved if all child-goals of this proof step are solved.

This means if all goals which have been created by one proof step are solved then the goal to which this proof step has been applied is solved too.

Procedure GoalDisproved:

The parent-goal of a disproved goal is disproved if the proof step was a

- *case analysis;*
- conclusion of a *lemma* or *hypothesis application* with empty or successful conditions;
- conclusion of a "lazy" *conditional rewrite step* with successful conditions.

If the conditions are not empty or successful for the last two points then all active goals in the subtrees of the conditions are stopped by the TIP system because their evaluation is not meaningful any more.

Also the siblings of a disproved case of a case analysis are stopped to avoid superfluous calculations.

The parent-goal of a disproved goal is not disproved if the proof step was a *generalization* or a *twin step*, because the generalized goal may be wrong without implying something about the correctness of its parent-goal.

The Order of Applying Proof Steps

Of course the application of the proof steps is not really done in parallel. Therefore the most interesting topic is the order in which the different proof steps are applied.

- The first goal of the "wave" queue becomes the "current" goal. (If the queue is empty then a warning is printed and the command interpreter is called.)
- If the "current" goal is not yet simplified then it is simplified now.
- Apply conditional rewrite rules and simplify the new child goals.
- Apply induction hypotheses and simplify the new child goals.
- Apply lemmas and simplify the new child goals.
- Apply "twin-steps" and simplify the new child goals.
- Generalize subterms and simplify the new child goals.
- Do case analyses and simplify the new child goals.
- Check if the theorem is solved or disproved and leave the proof algorithm if so.
- Call the command interpreter for user interaction.
- Remove the current goal from the "wave" and continue with the first point of this list again.

Thus, we have a loop which is only stopped when the theorem is solved or disproved or if the user enters the **EXIT** command in the command interpreter. The call of the command interpreter is skipped forever in this loop if the user enters the **GO** command or for *n* cycles if the **STEP *n*** command is used.

The order of the proof steps in the list above has been chosen in this way to try the most promising steps first. Therefore the child goals of these proof steps will be located in the queue before the rest of the new child goals. The single proof steps are influenced by the setting of some according system parameters as described in section 4.3. Moreover, some build-in heuristics (see the following section) of the TIP system may evade the application of superfluous proof steps.

4.6.3 Heuristics

The general idea of heuristics in the TIP system is to avoid unnecessary branches in the proof tree and to keep the proof tree slim. In difference to the Boyer-Moore prover [BM79] there are no heuristics implemented for selecting the "best" proof step for the continuation of a goal.

The following heuristics are implemented in the TIP system up to now:

- Evade the *inverse* application of induction hypotheses or lemmas for the same redex, because equations can be applied in both directions.
- Evade the repeated application of *permutative* induction hypotheses or lemmas for the same redex. Otherwise equations like $f(x, y, z) = f(y, z, x)$ would lead to loops.
- Detect and skip repeated applications of the same proof step which only *blow up* the goal without causing any progress. This is check by comparing the last n proof steps in a branch. If these are equal then it is tested if the last n+1 goals equal two by two except for some new inner "sub-term". If this sub-term is added with every proof step than we have detected a useless non-terminating branch in the proof tree which should be skipped.
  Let G be a goal of the form C[t] which consists of a context C[…] and a term t. If G has a child goal G' of the form C[c[t]] (where c[…] is another context) and G' again has a child goal of the form C[c[c[t]]] and if this continues till the n-th child of G with the form C[c^n[t]] then the blow-up check

skips this branch. The parameter *n* of the blow-up check can be selected with the system parameter **CYCLE_LIMIT** (see page 54). Its default value for automatic theorem proving is 2.

- Evade equivalent case analysis in different branches of the proof tree. This can be selected with the value **REDUCED_CA** for the system parameter **CUT_SELECTION** (see page 54).
- Skip the search for possible proof steps after the first applied proof step. This is only useful for interactive proving because the user can select the "best" possible continuation. This can be selected with the value **SINGLE_STEP** for the system parameter **CUT_SELECTION** (see page 54).

We are planning to implement some more heuristics. Very useful would be heuristics for selecting promising case analyses and generalizations. Besides, some kind of lemma-suggesting procedure would be a good idea.

4.6.4 Automatic and Interactive Prove Mode

There are two shorthand commands for setting the most important system parameter to useful values for interactive proving (**SET PARAM INTERACTIVE**) and for automatic proving (**SET PARAM AUTOMATIC**). From now on we will refer to these setting as *interactive* and *automatic mode*. If you start the TIP system then you are in the automatic mode.

Normally in the automatic mode you just type the **GO** command and then you wait for the system message:

```
# THEOREM NO. 1 PROVED BY SYSTEM.
```

If this message does not appear then you can kill the system by the UNIX interrupt Control-C or wait for the "ran out of memory" error message. To avoid this it may be better to use the **STEP 50** command. Here the system will stop in the command interpreter after 50 cycles of the proof algorithm. Then you can have a look at the proof tree and/or the queue of active goals. You can try another **STEP *n*** command but probably the TIP system needs some help to prove this theorem (if possible at all!).

A good use of the automatic mode is the search for the shortest proof if you know that TIP can solve this theorem. Because of the breadth-first-search in automatic mode TIP will always find those proofs which need the fewest proof steps. On the other hand you will run in time or space problems in this mode very often.

The interactive mode is designed to solve theorems in a cooperation with the user. TIP will suggest all possible proof steps and the user can select the "best" one. Moreover, in this mode it is easy to compare why the induction hypothesis does not "fit" to a goal. The most common reasons for the failure of an inductive proof are that a goal is too special or that some necessary information is missing. The first one sometimes can be solved by a generalization, the second one by entering lemmas (maybe with an assumed correctness).

In both modes the rewriting is done automatically. This is useful for most of the proofs, but there are some situations where you even may want to have interactive rewriting. For example if you want to use a lemma in a direction which is "against" the rewrite rules then this lemma application would be undone immediately by the automatic rewriting. Another reason for switching off the automatic rewriting may be the preservation of special (constructor-)terms. For example if you want to use a lemma which needs the number '1' as argument and rewriting always transforms this to zero and some new context then doing the rewrite steps interactive may be necessary.

4.7 Proof Output

If you and/or the TIP system finally finished a proof you may be interested in a pretty-printed listing of the complete proof. Especially if the proof was done automatically by the system this proof output is important.

To get this output you have to use the **SHOW PROOF** command. This command cannot be used as long as the proof is not finished. If you add the option **/FILE** then the proof listing will be written to a file.

This output consists of a step-by-step description of the proof and some brief statistics. There are a lot of options for selecting the desired form of output. Please see section 4.9 for information about these options. The most important ones are **/COMPRESS** for combining several rewrite steps in every line of the proof output and **/UFORM** for a different form of output. Here one starts with the left hand side of an equation and transforms it step by step to its right hand side.

If a proof has needed one or several lemmas then the proofs of these lemmas are listed in front of the proof for the theorem. Or, if the correctness of the lemma was assumed by the user then this is mentioned too at the beginning of the proof output.

4.7.1 Listing

If a theorem was proved by TIP then you can enter the **SHOW PROOF** command to get a proof listing of the following form:

```
TIP/PROOF> show proof

TIP System Version 3.0 SUN 4.1/UNIX Listing_of_Proof


PROOF OF THEOREM

 rev(rev(X1)) = X1

 COMPLETE CASE ANALYSIS on X1                                              => [1]
 (Generated from the rules for function rev)

1. CASE: X1 = empty

   - rev(rev(empty)) = empty
   - rev(empty) = empty                                                 (by REV.1)
   - empty = empty                                                      (by REV.1)
   CASE 1. finished

2. CASE: X1 = append(X2,X3)

   - rev(rev(append(X2,X3))) = append(X2,X3)
   - rev(conc(rev(X3),append(X2,empty))) = append(X2,X3)                (by REV.2)
   - rev(conc(rev(X3),append(X2,empty))) = append(X2,rev(rev(X3)))
                                                                  (by IND.HYP.[1])
   - rev(conc(X4,append(X2,empty))) = append(X2,rev(X4))      (by GENERALIZATION)


   COMPLETE CASE ANALYSIS on X4                                            => [2]
   (Generated from the rules for function conc)

   2. 1. CASE: X4 = empty

      - rev(conc(empty,append(X2,empty))) = append(X2,rev(empty))
      - rev(append(X2,empty)) = append(X2,rev(empty))                   (by SEQ.5)
      - conc(rev(empty),append(X2,empty)) = append(X2,rev(empty))       (by REV.2)
      - conc(empty,append(X2,empty)) = append(X2,rev(empty))            (by REV.1)
      - append(X2,empty) = append(X2,rev(empty))                        (by SEQ.5)
      - X2 = X2
        & empty = rev(empty)                                      (append removed)
      - empty = empty                                                   (by REV.1)
      CASE 2.1. finished.

   2. 2. CASE: X4 = append(X10,X11)

      - rev(conc(append(X10,X11),append(X2,empty))) =
          append(X2,rev(append(X10,X11)))
      - rev(append(X10,conc(X11,append(X2,empty)))) =
          append(X2,rev(append(X10,X11)))                               (by SEQ.6)
      - conc(rev(conc(X11,append(X2,empty))),append(X10,empty)) =
          append(X2,rev(append(X10,X11)))                               (by REV.2)
      - conc(rev(conc(X11,append(X2,empty))),append(X10,empty)) =
          append(X2,conc(rev(X11),append(X10,empty)))                   (by REV.2)
      - conc(append(X2,rev(X11)),append(X10,empty)) =
          append(X2,conc(rev(X11),append(X10,empty)))             (by IND.HYP.[2])
      - append(X2,conc(rev(X11),append(X10,empty))) =
          append(X2,conc(rev(X11),append(X10,empty)))                   (by SEQ.6)
      CASE 2.2. finished.

THEOREM PROVED
```

```
STATISTICS                          THEOREM
  Lemma applications        :          0
  Generalizations           :          1
  Induction hypotheses used :          2
  Number of case analyses   :          2
  Total number of cases     :          4
  Decomposition steps       :          1
  Rewrite rule applications :         12
  CPU-time used (in msec)   :        817
```

A "normal" line starts with a minus sign (`-`) then an equation is listed and on the right end of the line TIP writes the rewrite or proof step which was used to come from the equation in the line above to the equation in this line. If a goal consists of several equations then the following line(s) start with an ampersand (`&`) to show that several equations belong together. Mainly this happens after decomposition steps which may split one equation into several smaller ones.

When a case analysis is done then TIP writes "**COMPLETE CASE ANALYSIS** `on`" and states all variables which have been substituted in one of the cases. Furthermore, on the right end of this line the number of the induction hypothesis is written which was created for this case analysis. If later in the proof the statement "`by` **IND.HYP.[1]**" appears then it means that an instance of the induction hypothesis `[1]` has been used. The information `=> [1]` denotes that the induction hypothesis [1] was generated from the goal above this line.

Every case of a case analysis is marked by a number and its substitution. The end of each case is shown by "`CASE` *n*`. finished`". If case analyses are nested then the cases are numbered 2.1, 2.2 and so on.

Possible information about the creation of a goal can be:

`- (by TYPE.2)`"	for a rewrite step with this axiom,
`- (by` **LEMMA No. 2**`)`"	for the application of a lemma,
`- (by` **GENERALIZATION**`)`	if the goal is a generalization of the goal above,
`- (by` **IND.HYP.[2]**`)`	for the application of an induction hypothesis,
`- (`*function* `removed)`	for a decomposition step for this function symbol,
`- (by AC-equal)`	if both sides are equal modulo AC.

The important proof steps are printed boldface. Together with the case analyses these steps form the structure of an inductive proof.

4.7.2 Statistics

At the bottom of a proof listing the TIP system gives some statistical information about the proof.

```
STATISTICS                         LEMMA No. 1   LEMMA No. 2      THEOREM
 Lemma applications        :             0             2            1
 Generalizations           :             0             0            0
 Induction hypotheses used:              0             0            1
 Number of case analyses   :             0             0            1
 Total number of cases     :             0             0            2
 Decomposition steps       :             1             0            1
 Rewrite rule applications:              2             1            4
 CPU-time used (in msec)   :            29            84          217
```

These are mainly how many proof steps of every sort have been used and how much time has been needed. Moreover, the sum of the number of cases of all case analyses is given. This value counts all cases, sub-cases and so on. If you divide this number through the number of case analyses then you have a kind of measure for the complexity of the case analyses. If this value is 2 then there have been only "simple" case analyses. If the value is higher than 2 there have been some more "sophisticated" case analyses.

We used the CPU-time (in milli seconds) as measurement for the duration of a proof to be independent from the pace of the user. If auxiliary lemmas have been proved then their statistics are also printed too.

If you do not want to get the statistics then you have to use the **/NOSTATISTIC** option.

4.8 Dealing With Lemmas

For non-trivial proofs the TIP system normally needs help. This help usually is given in the form of lemmas. These lemmas may state new information about the used data structures. The need of lemmas is obvious when TIP is not able to apply the induction hypothesis in the induction case. In this case you should try to analyze what kind of "axiom" is missing. This can be done by comparing the hypothesis with the goal.

We are planning to implement a kind of support tool for this comparison of goals and left hand sides of hypotheses. But of course this is not trivial because you may have to look at several possible application redices, consider both directions of the hypothesis and check the fulfillment of the induction ordering.

Lemmas are given to the system by the **ENTER LEMMA** command. If you do not want to prove this new lemma before using it you should use the **/ASSUMED** option. Otherwise TIP will start a "parallel" proof of this new lemma, i.e. the other proof is suspended until the current proof is finished. It is also possible to enter lemmas again for the proof of a lemma. TIP uses a kind of queue to manage this simultaneous working on several proofs. If you want to view and influence this order then you can use the **SHOW QUEUE THEOREMS** and the **SET CURRENT THEOREM <theorem-number>** commands. It is not possible to "remove" an incomplete proof, but bringing other proofs to the front of the queue will have the same effect.

For changing from one proof to another TIP just has to store up the tree and the goal queues of the "old" proof. This information is enough to restore a proof whenever it should be continued. The goals of all proofs are numbered consecutively. Besides, the system variables are unique over all proofs. Thus, there can be no conflict with matching goal or variable numbers when changing the current proof.

Like induction hypotheses a lemma is a non-directed (conditional) equation and therefore it can be used in both directions. Both proof steps look quite similar to the user especially because the decisive difference is not visible. For the application of a lemma no ordering has to be checked before presenting this rule to the user (interactive mode) or applying it (automatic mode).

There is no real difference between the proof of the theorem and of any lemma. The theorem was used to start the proof session and therefore its name will be used always for naming some files like proof listing or trace.

4.9 Commands in PROVE Mode

The commands described in this section are available only in PROVE mode (prompt "`TIP/PROVE>`"). Some of the TIP commands described in section 3.5.1 are valid in PROVE mode, too. Not usable in PROVE mode are the **PROVE** command itself, the **THEOREMEDIT** command, the **UNIFY** and the **MATCH** commands and the **COMPLETE** command because these commands would change to different "states" of the TIP system. Furthermore, the **SET FUNCT** commands, the **SET PARAM ORDERING** and the **ASSUME** command are not allowed in PROVE mode because these commands would change that part of the proof environment which has to be "stable" during a proof session for a correct result.

On the other hand many "new" commands are available in PROVE mode for inspecting and controlling the current state of a proof. Of course the command completion, the question mark ('`?`') and the **HELP** command work quite the same way as in STANDARD mode.

STEP or **STEP *n***

Proceed one or more proof steps.

A **STEP** command without an argument allows the proof algorithm to perform the next step. If a number n is given as argument then the next n proof steps are done. This is synonymous with entering the **STEP** command *n* times.

The **STEP** command is the default command for the PROVE mode, i.e. it is performed for empty input (only RETURN key is pressed).

Parameter: If a number n is supplied as an argument, the algorithm performs n proof steps without returning to the interactive command input. This is not very useful if some system parameters are switched to their interactive values!

GO

Continue the proof without prompting for user commands.

The **GO** command terminates the interactive PROVE mode until a proof for the current lemma or theorem is found. This is not very useful if some system parameters are switched to their interactive values!

SHOW GOAL

Display goals.

This command serves for the inspection of goals during the proof of a theorem.

The goal can be selected by an argument. If no argument is given, the actual goal is shown.

Parameter: The (optional) argument of a **SHOW GOAL** command has to be the number of a goal which has been created within the current PROVE session.

SHOW GOAL HYPOTHESIS

List all induction hypotheses which are available for a goal.

This command is used to check which induction hypotheses could be applied to a goal. All hypotheses which have been generated from any parent of this goal are listed. The goal can be selected by an argument. If no argument is given, the actual goal is used as argument.

Parameter: The (optional) argument of a **SHOW GOAL HYPOTHESIS** command has to be the number of a goal which has been created within the current PROVE session.

SHOW TREE

Display the structure of the proof tree.

A **SHOW TREE** command results in a display of the goals generated within the current PROVE session (represented by the goal numbers). The tree structure of these goals is shown by indentation: If a goal g2 has been produced by a proof step from goal g1, then g2 is written below g1, indented by three spaces. The goal numbers are labelled with the current status of a goal and the goals are preceded by the information about the proof step they have been generated from.

The current text of the goals themselves can be inspected by a **SHOW GOAL** command.

Options:	**/FILE /NOFILE /GRAPHED**	(Default: **/NOFILE**)
	/ACTUAL /FULL	(Default: **/ACTUAL**)

/FILE Write the proof tree to a file instead of the terminal. The name of the file is the name of the type with the extension `.prt`.

/NOFILE Show the listing on the terminal only.

/GRAPHED Write the proof tree in a very special format to a file. The name of the file is the name of the type with the extension `.gra`. This file can be used as input for the graph editor system *GraphEd*[1].

/ACTUAL Show only the child-goals of active goals. That means that successful or failed branches of the proof tree are only displayed by their uppermost goal node. This option is useful if you want to see which parts of your proof are still “unsolved”.

/FULL Show the whole proof tree without cutting any branches. This option is useful if you want to see the proof tree of a proved theorem. Normally such a tree would only be displayed as the solved start-goal.

Parameter: The (optional) argument of a **SHOW TREE** command has to be the number of a goal

1. This system is available via *anonymous ftp* from forwiss.uni-passau.de. For more information please contact M. Himsolt with the e-mail address graphed@informatik.uni-passau.de.

which has been created within the current PROVE session. If a goal number is given as argument then only the sub-tree below this node of the proof tree is displayed. This may be very useful if you are only interested in a small part of a very large proof tree. If no argument is given then always the total proof tree is displayed.

SHOW PROOF

Display a proof listing of a completed proof.

When a proof has been successfully finished (Message: `# THEOREM NO. x PROVED BY SYSTEM.`) a pretty-printed proof listing of the theorem with the number *x* can be displayed on screen or printed to a file. A **SHOW PROOF** command results in a proof output and some statistics for the current theorem or for the lemma which is specified as argument. There are many options available for controlling the form of the proof output.

Options:	**/FILE /LAFILE /NOFILE**	(Default: **/NOFILE**)
	/BRIEF /STANDARD /DETAILED	(Default: **/DETAILED**)
	/LINEAR /UFORM	(Default: **/LINEAR**)
	/COMPRESS /NOCOMPRESS	(Default: **/NOCOMPRESS**)
	/STATISTIC /NOSTATISTIC	(Default: **/STATISTIC**)
	/BLOCKMAX = <integer>	(Default: **5**)
	/REWRITEMIN = <integer>	(Default: **5**)

/FILE	Write the proof listing to a file instead of the terminal. The name of the file is the name of the type with the extension `.prf`.
/LAFILE	Write the proof listing in the form of a LaTeX file. The name of the file is the name of the type with the extension `.tex`. Thus, a proof listing can easily included in LaTeX reports or other documents.
/NOFILE	Show the listing on the terminal only.
/BRIEF	Only the structure of the proof is listed.
/STANDARD	The steps during the simplification of goals are not displayed.
/DETAILED	The goals are displayed after every proof step (including every simplification step).
/LINEAR	The listing is printed with the whole equation(s) in every line.
/UFORM	The proof listing starts with the left hand side of an equation and transforms it successively into the right hand side (as far as this is possible for the proof structure).
/COMPRESS	Group several (non-ramifying) proof steps into one line. The amount of grouped steps depends on the option values for **/BLOCKMAX** and **/REWRITEMIN**.
/NOCOMPRESS	Do not group any proof steps.
/STATISTIC	Give some statistics about the proof at the end of the listing.
/NOSTATISTIC	Do not give the statistics about the proof.
/BLOCKMAX	The value of this option defines how many groups of proof steps should be at most in one "block", i.e. between two ramifications or after the last ramification in every branch of the proof tree. This value is only relevant if the **/COMPRESS** option is used. The default value is 5.
/REWRITEMIN	The value of this option defines how many proof steps should be grouped in one line of proof output at least. This value is only relevant if the **/COMPRESS** option is used. The default value is 5.
Parameter:	The (optional) argument of a **SHOW PROOF** command has to be the number of a lemma or theorem which has been proved within the current PROVE session. If no argument is given then always the current proof is listed.

SHOW QUEUE or **SHOW QUEUE GOALS**

Display the queue of active goals.

A **SHOW QUEUE GOALS** command displays the queue of active goals, called wave. These goals are "waiting" to be processed by the proof algorithm. The goals in this queue will be processed left to right. The goals are represented by goal numbers.

The current text of the goals can be inspected by a **SHOW GOAL** command, the tree structure of the goals is obtained by a **SHOW TREE** command. The order of the queue can be changed with the **SET CURRENT** command.

SHOW QUEUE [KEEP | SOLVED | STOPPED]

Display the queue of non-active, solved or stopped goals.

A **SHOW QUEUE [KEEP | SOLVED | STOPPED**] command displays the queue of non-active, solved or stopped goals. The goals are represented by goal numbers.

In the "keep" queue are those goals which have already been processed by the proof algorithm and which are neither solved nor disproved. So, they have to wait until some new information is inherited from their child-goals or until they are stopped because a "parallel" branch of the proof tree was successful.

In the "solved" queue are those goals which have been trivial after the simplification or which have inherited the success information from their child-goals.

In the "stopped" queue are those goals which have been stopped by the TIP system, because they are not needed any more.

SHOW QUEUE THEOREMS

Display the list of theorems.

A **SHOW QUEUE THEOREM** command displays the list of theorems and lemmas which are active in the current PROVE session.

The theorem or lemma itself can be inspected by a **SHOW THEOREM** command.

Options:	**/ACTUAL** **/DONE**	(Default: **/ACTUAL**)
/ACTUAL	Show the list of unfinished proofs, i.e. proofs which are still active.	
/DONE	Show the list of finished proofs, i.e proved or disproved theorems.	

SHOW HYPOTHESIS

Display induction hypothesis.

This command shows the text of an induction hypothesis. The list of valid hypotheses for a goal can be inspected with the **SHOW GOAL HYPOTHESIS** command.

Parameter: The argument of a **SHOW HYPOTHESIS** command has to be the number of an induction hypothesis which has been created within the current PROVE session.

SHOW THEOREM

Display lemma or theorem.

This command serves for inspection of lemmas or theorems during the execution of an induction proof.

Parameter: The argument of a **SHOW THEOREM** command has to be the number of a lemma which is available within the current PROVE session.

SHOW RULE

Show rewrite rules (axioms).

This command looks for the defining text of rewrite rules. It needs as an argument

- either a name of a type (displaying all axioms of this type),
- or a name of an axiom (displaying only one axiom).

Parameter: An argument for **SHOW RULE** may be:

- a type identifier known in the current TIP session,
- a pair of a known type identifier and an axiom name.
  Such pairs have to be written as **<typename>.<axiomname>**, i.e. separated by a dot and without any blank characters. For example `NAT.3` would be a correct argument.

ENTER LEMMA

Make a new lemma available in the current proof session.

An **ENTER LEMMA** command is used if the user recognizes the need of a new lemma during a PROVE session. The new lemma can either be an existing theorem or an equation which is entered interactively (see section 4.3.4 and section 5.1). If the new lemma is entered without assuming its correctness then the TIP system will suspend the current proof and start the proof of the new theorem. If this proof is finished the current proof will be continued.

Options: **/ASSUMED** **/UNPROVED** (Default: **/UNPROVED**)

/ASSUMED The correctness of the new lemma is assumed by the user. The lemma is effective immediately and can be used in other proofs.

/UNPROVED The correctness of the new lemma has to be proved by the TIP system. Only after the proof of the lemma it can be used in other proofs.

Parameter: The (optional) argument of an **ENTER LEMMA** command has to be a name of a known theorem. A list of all known type and theorem names can be obtained by the **DIR** command. If no argument is given then the new lemma has to be entered interactively.

ENTER CONDITION

Enrich the conditions with a new term and do a case analysis for this term.

An **ENTER CONDITION** command asks the user to enter a term. From this term and a new variable an equation is built and appended to the conditions of the goal (specified as argument). Then a case analysis is done for this new variable. So, we can do a complete case analysis for an arbitrary term. This may be very useful if the term blocks the application of rewrite rules or if it is a necessary information for the application of conditional rules (see section 4.3.2).

The term may only contain system variable which occur in the goal. So, all variables must have the form 'X*n*' where *n* is a natural number. The term is entered interactively (see section 5.1).

Parameter: The (optional) argument of an **ENTER CONDITION** command has to be the number of a goal contained in the current goal queue (see **SHOW QUEUE**). If no argument is given the current goal will be used.

GENERALIZE

Generalize one or several subterms.

A **GENERALIZE** command starts the generalization algorithm for the current goal or the argument goal. This is similar to the generalization proof step (see section 4.3.5). This command is useful for proofs where generalization is switched off and should be done only by command. The options are similar to the argument values of **SET PARAM GENERALIZATION** command.

Options: **/ALL** **/MULTIPLEONLY** **/INTERMULT** **/INTERACTIVE**
(Default: **/INTERMULT**)

/ALL Generalize all possible subterms.

/MULTIPLEONLY Generalize only such terms which appear several times in the goal.

/INTERMULT Generalize in an interactive way only such terms which appear several times in the goal.

/INTERACTIVE Do all possible generalizations in an interactive way.

Parameter: The argument of a **GENERALIZE** command has to be the number of a goal contained in the current goal queue (see **SHOW QUEUE**). If no argument is given the current goal will be used.

SET CURRENT or **SET CURRENT GOAL**

Select the next goal to be processed by the proof algorithm.

A **SET CURRENT** command rearranges the goal queue by putting the goal given to it as an argument on top of the queue. This way it redefines the continuation point of the algorithm, since the next "current goal" will be taken from the top of the queue, so the goal given to **SET CURRENT** will be the next one.

Parameter: The argument of a **SET CURRENT** command has to be the number of a goal contained in the current goal queue (see **SHOW QUEUE**).

STOP GOAL

Remove a goal from the goal queue.

The **STOP GOAL** command serves for removing goals from the goal queue which are decided by the user not to be treated further. The stopping of a goal can be undone by an **ACTIVATE GOAL** command. Another way for influencing the search is **SET CURRENT**.

Parameter: The argument of a **STOP GOAL** command has to be the number of a goal contained in the current goal queue (see **SHOW QUEUE**).

ACTIVATE GOAL

Put a stopped goal back into the goal queue.

The **ACTIVATE GOAL** command serves for moving a stopped goal back into the goal queue.

Parameter: The argument of an **ACTIVATE GOAL** command has to be the number of a goal contained in the queue of the stopped goals (see **SHOW QUEUE STOPPED**).

SET CURRENT THEOREM

Select the current theorem.

A **SET CURRENT THEOREM** command rearranges the theorem queue by putting the theorem given to it as an argument on top of the queue. By this way the current proof may be moved to the 'background' and the proof algorithm deals with the new theorem.

Parameter: The argument of a **SET CURRENT THEOREM** command has to be the number of a lemma or theorem contained in the current theorem queue (see **SHOW QUEUE THEOREMS**).

DELETE LEMMA

Remove a lemma from the theorem queue.

The **DELETE LEMMA** command serves for removing lemmas from the theorem queue which are decided by the user not to be treated further. Note that there is no way for undoing a **DELETE LEMMA** command, so the safer way for influencing the search is **SET CURRENT THEOREM**.

Parameter: The argument of a **DELETE LEMMA** command has to be the number of a lemma or theorem contained in the theorem queue (see **SHOW QUEUE THEOREMS**).

EXIT

Leave the PROVE mode.

If the **EXIT** command is used in the PROVE mode it switches back to the STANDARD mode. A 'logical end-of-file' (Control-D) is equivalent to **EXIT**.

4.10 Useful Hints

In this section we want to collect some points which seem to be helpful for proving theorems with the TIP system. After several hundred of proof sessions we have some experience with interactive proving.

Maybe some of the following hints are useful for users who want to deal with the TIP system:

- If you have a theorem and you think this could be proved in an automatic way but if you are not really sure, then it is no good idea to use the **GO** command. It is better to use the **STEP *n*** command with an argument like 50. In this case the TIP system will deal with the first 50 goals and then return to the command interpreter. This gives you the possibility to have a look at the goal queue and the proof tree. Then you can decide if you want to try another **STEP *n*** command (use a smaller argument this time, because the goals are larger now and every step needs a longer time) or if you stop this proof. The GO command is a little like "all or nothing" and "nothing" means a long time of waiting, a "ran out of memory” message and a large core dump. Even killing TIP and the current proof session with a CTRL-C interrupt and starting the system again is a better solution.

- If you try to prove a theorem in the interactive way then you should always keep in mind that the application(s) of the induction hypothesis is the most promising step if you deal with recursive constructor functions. Therefore, after doing a case analysis, you should always compare the goal of the “recursive” case and the induction hypothesis. This can be done with the **SHOW GOAL <goal-no>** and the **SHOW HYPOTHESIS <hyp-no>** commands. If the hypothesis does not fit then you should try to find out the reason. Maybe another case analysis is necessary or a lemma could be used to transform the goal into a form so that the hypothesis would fit.
- If you think that a hypothesis should fit to a goal, but the TIP system does not propose this step then normally the system is right and something prevents the application of the hypothesis. First check if the hypothesis would fit syntactically. Look carefully that there are no different constants (ground terms) like ‘1’ and ‘0’ which prevent a match. Another reason for a failing match could be a variable which appears several times in the hypothesis. In this case every occurrence of this variable must be substituted by exactly the same term. To give a simple example: A hypothesis of the form $f(y, y) = 0$ cannot be applied to a goal of the form $f(x, \mathrm{succ}(x))$, because different terms would be substituted to the variable y.
- Try to keep a picture of the proof tree in your mind. If you do an interactive proof you should always know in which branch you currently work. Then you know which induction hypotheses are available here and which proof steps have been applied to reach the current goal. Otherwise you lose the control of your proof very quick and then you only can guess whether a proposed proof step is useful or not. Normally it is a good idea to use the **SHOW TREE** command every time you finish a branch.
- One of the most difficult jobs of the user is to recognize the need of a lemma and to find its “best” form. Often it is easier to proof a more general form of a necessary lemma, e.g. replacing constants by variables. Use the **ENTER LEMMA** command to state the new lemma. If your lemma turns out to be “unprovable” then you can use the **SET CURRENT THEOREM 1** command to switch back to the main proof.
- If it seems that you decided to do a “wrong” proof step in your interactive proof session then do not be to stubborn. Do a backtracking by hand. Look at the proof tree and find out the goal where another continuation might be more promising. Then use the **SET CURRENT <goal-no>** command to go back to this goal and try a different proof step.
- Automatic rewriting to normal form is usually a good idea. Therefore it is used in automatic as well as in interactive prove mode. But there may either be theorem with constants where you need to keep some pattern for the application of the induction hypothesis. If this pattern would be destroyed by automatic rewriting then you use interactive rewriting. Another reason for interactive rewriting are the application of lemmas which perform a step against the rewrite direction. Here you also have to avoid automatic rewriting, because otherwise the lemma application would be ‘un-done’ by automatic rewriting immediately.
- Even though there are heuristics implemented which can detect simple cycles in the proof tree, you should nevertheless also try to make out cycles. Cycles are often a sign that a lemma is necessary to solve this prove.

5 Editing Theorems

One useful feature of TIP is the possibility of creating or editing theorems without leaving the system. Sometimes one would like to have a slightly different version of an existing theorem if the theorem itself is stuck at a special point. Or, one has the idea of a useful lemma and one would like to generate and try to prove it. Normally one would have to “leave” the TIP system and create or change an input file with the help of a UNIX editor. Then one would have to use the **READ** command or to start TIP again. Also this way of editing theorems may often cause syntax errors which force the user to check the input file again.

To make it easier to deal with theorems there is a theorem editor included in the TIP system. This editor allows the user to change existing theorems or to generate new theorems. This editor is started with the **THEOREMEDIT** command. The argument of this command has to be an existing theorem name or a “new” name, i.e the name may not be the name of any existing type.

While editing a theorem you are in the EDIT mode of the TIP system. This is indicated by a different prompt (‘**>>**’). In the EDIT mode there exists always a current equation. This equation is displayed in the line above the prompt. The display of this equation also contains information about the sort which is necessary to continue this equation. The sort information is enclosed in question marks. For example

```
(1): add(x,?Nat?) = *
>>
```

means that you are about to enter the first equation. The standard function `add` and a variable `x` have already been entered. Now TIP indicates that the next input for the equation has to be a natural number (“`?Nat?`”). This is possible because an on-line check of syntax and context conditions is performed for the user input immediately. The missing right side of the equation is symbolized by an asterisk.

To give TIP the possibility to distinguish between commands and equation input every command has to be prefixed by a *dot* as *escape character*, e.g. to use the **DIR** command you have to type “`.dir`”. Every input which does not start with a dot is interpreted as a function or variable symbol.

The next section explains how to enter functions and variables for generating new (unconditional) equations. Then the commands of the EDIT mode are summarized. In sections 5.3 you can learn how to edit existing equations and in the last section the handling of conditional equations is explained.

5.1 Equation Input

The input for equations is checked in a syntax directed way. This means that TIP recognizes if you typed a known function or something else. Everything which is no function will be interpreted as a variable. Structuring characters like brackets, commas and equation symbols are treated like blanks because the structure of the terms is calculated by the TIP system itself. For example the following “wrong structured”[1] input would lead to a correct equation:

```
(1): ?left side? = *
>> add(x), y) =)add((=, y    x ,,(

###> `x` was declared as variable of sort Nat
###> `y` was declared as variable of sort Nat
Equation Nr. 1 created:
(1): add(x,y) = add(y,x)
```

As shown above you can enter a whole term or equation in one line, e.g. by the put and get operations of your window system, but you can also enter one symbol or variable each time and then press <RETURN> to see the necessary sort of the next input symbol. The second is very useful if you are not sure about the correct order of the arguments of a function.

1. Brackets, equation symbols and commas are used in an incorrect way here. The function symbols and variables have to be in the correct order to form the desired equation.

The editor also manages the lists of necessary “based on” types and of variable declarations. If you introduce or delete a function symbol or a variable one of these lists may change. The current theorem can always be viewed by the **.LIST** command (without using an argument):

```
>> .list

TEMPORARY theorem DEMO

   basedon NAT

   unknown x, y: Nat

   goals

(1): add(x,y) = add(y,x)

(2): ?left side? = *
>>
```

For creating a new theorem only the functions and variables which build up the equation(s) have to be entered. The “rest” of the theorem is created by the theorem editor.

For an empty equation the editor shows the equation number and the string `left side` enclosed in questions marks, i.e. the sort of the left side is not known up to now. If a variable is entered for the left hand side of the equation then it is declared as a variable of sort “`yet_UNKNOWN`”. It will be redeclared as soon as its sort is decidable. Therefore equations with a single variable on every side are not allowed, because in this case the sort of these variables would be not decidable.

```
(2): ?left side? = *
>> z
###> `z` was declared as variable of sort >yet_UNKNOWN<
(2): z = ?right side?
>> or
###> `z` was declared as variable of sort Bool
(2): z = or(?Bool?, )
>>
```

If in the EDIT mode a string is entered which does not start with a dot (‘`.`’) then the following points are checked:

- If the string is a minus character (‘`-`’) then the last symbol of the equation is deleted. It is checked too if any implicit declaration (for variables or “based on” types) is unnecessary now.
- If the string is a question mark (‘`?`’) then a short information about possible input is printed (see also section 5.2).
- If the string is a known function symbol, i.e. it is declared as a constructor or function in any type currently loaded in TIP, then the declaring type is added to the list of “based on” types if not already there. If the function symbol is used in several different types then the user is asked for a decision.
- Otherwise the string is interpreted as a variable. If this is the first occurrence of the variable then it is declared and a message is printed to the screen. Otherwise it is checked if the current use of the variable is correct according to the sort information stored for this variable. If not, an error message is printed:

  ```
  (2): z = or(?Bool?, )

  >> x

  ------ERROR: SORT CONFLICT

        Sort Nat found where sort Bool is needed!
  ```

5.2 Commands of the EDIT Mode

All command which are used in the EDIT mode have to start with a dot as escape character. Otherwise the command would be interpreted as input for the current equation. Only the **THEOREMEDIT** command does not need this escape character because it is used in the STANDARD mode to invoke the EDIT mode.

THEOREMEDIT

Context supported editing of theorems.

An existing theorem can be edited or a new theorem can be created with the **THEOREMEDIT** command. Enter '**?**' in EDIT mode to get a complete list of all possible editor commands. All editor commands have to start with a dot, otherwise they will be treated as input for equations! There is no need for brackets or other special separator signs because the editor is strongly context supported. Implicit variable declarations are indicated.

This command is only available in the STANDARD mode of the TIP system.

Parameter: The argument can be the name of an existing theorem or any other string except the name of an existing type.

.LIST

Display the defining text of a type or theorem.

Given a known type or theorem name as argument, this command displays the textual representation of the type or theorem in a pretty-printed manner. If the **LIST** command is used without an argument the current "structure" of the currently edited theorem is displayed in a pretty-printed manner.

For possible options and parameter values see **LIST** command in section 3.5.

.DIR

Directory of known types and theorems.

A list of all known type and theorem names is displayed, i.e. a list of all identifiers for which a type or theorem declaration has been read from the input file and additional the two standard types `BOOL` and `NAT`.

For possible options and parameter values see **DIR** command in section 3.5.

.CLEAR

Delete one equation.

The **CLEAR** command is used to delete one existing equation from the theorem or if used without an argument it aborts the presently unfinished equation.

Parameter: If an argument is used then it has to be the number of an existing equation.

.COPY

Make a copy of an existing equation.

The **COPY** command is used to generate a copy of an existing equation. This copy can than be changed without influencing the original equation. The new equation is stored under the next free equation number[1]. A list of all equations including their numbers can be obtained by the **.LIST** command.

Parameter: The argument has to be the number of an existing equation.

.CHANGE

Make changes in an existing unconditional equation.

The **CHANGE** command is used to substitute a (sub)term of an existing equation by a new term of the same sort. If you are not familiar with the keys for selecting the substitution redex then press '`?`' when you are asked for the position. (In short: '`d`','`u`','`>`','`<`','`a`' will work.) To make another substitution on the same equation one has to use the **CHANGE** command again.

1. The equations are designated with the numbers 1 to *n*. The next free equation number then is *n+1*.

A more detailed description of this way of editing existing equations follows in section 5.3.

A list of all equations including their numbers can be obtained by the **LIST** command. This command can only be used if there is currently no unfinished equation[1] in the edited theorem.

Parameter: The argument has to be the number of an existing unconditional equation.

.SPLIT

Split an existing conditional equation.

The **SPLIT** command is used to cut a conditional equation up into its unconditional equational parts. I.e. every condition and the conclusion of this conditional equation will become one unconditional equation in the edited theorem. The split equation will be deleted. This is necessary because only unconditional can be edited with the **CHANGE** command. The **COMPOSE** command is used to build up the conditional equation again.

Parameter: The argument has to be the number of an existing conditional equation.

.COMPOSE

Build a conditional equation out of several existing equations.

The **COMPOSE** command is used to construct a conditional equation out of several unconditional parts. The numbers of those parts have to be given as argument of the command. First the numbers of the conditions and the last number has to be the equation which should become the conclusion. The parts are deleted and the new conditional equation is created instead. Therefore, if you need one of the parts for anything else than this conditional equation you should copy it first.

Parameter: The argument has to be a list of the numbers of at least two existing unconditional equations. The last equation in this list will be the conclusion, the others will be conditions.

.HELP or '?' (Question Mark)

Display information about TIP commands or list the possible commands respectively.

The **HELP** command invokes an interactive manual about the TIP command language (see section 3.3). Empty input (RETURN key) returns from help mode. The *question mark* ('?') prints a list of possible commands for the current state together with a short explanation. This works a little bit different to the question mark as described in section 3.3. For example if you enter a question mark while there is no unfinished equation you get the following output:

```
(1): ?left side? = *
>> ?
Possible input for theorem editor:
 ?                                                          to get this short help
 .LIST                                                       to list the equations
 .HELP                                              to get help about the commands
 .DIR                                               to list all types and theorems
 .LIST unitname                              to list the specified type or theorem
 .CLEAR {equ_nr}                      to remove one equation. If no number is used
                                        the current unfinished equation is removed
 .COPY equ_nr                             to make a copy of the specified equation
 .SPLIT equ_nr                                 to split a conditional equation
 .COMPOSE equ_nr {equ_nr} equ_nr        to build a conditional equation out of the
                                        first equations as conditions and the last
                                                                 as the conclusion
 .CHANGE equ_nr                                   to edit the specified equation
 .ABORT                                               to abort the whole theorem
 .EXIT                                   to save theorem and leave theorem editor
 Any variable, function or constructor with result sort arbitrary.
```

If there is an unfinished equation then the **CHANGE** and the **EXIT** commands are not available. On the other hand the minus sign ('-') (see below for a description) is a possible input now.

1. An equation is called *unfinished* if at least one symbol has been entered for this equation and the right hand term is not completely entered.

– (minus sign)

Remove the last function symbol or variable from an unfinished equation.

The minus sign ('-') can only be used while one has an unfinished equation. The input of a minus sign removes the last variable or function symbol of the current equation. If the last occurrence of a variable is deleted in this way then its declaration is removed too.

.ABORT

Abort the whole theorem editing process.

The **ABORT** command terminates the editing of a theorem and returns to STANDARD mode without saving the current theorem. If an existing theorem was edited it remains unchanged.

.EXIT

Save the theorem and leave the theorem editor.

The **EXIT** command can only be used if there is no unfinished equation in the current theorem. It saves the theorem under the given theorem name. If this name already exists a warning is printed:

```
WARNING: Theorem DEMO already exists!
Should it be overwritten by this new theorem? (y/n)
```

If the theorem should not be replaced then TIP asks for a new theorem name:

```
Please enter new theorem name or use CTRL-D to abort theorem.
THEOREMNAME>>
```

Now a new name for the edited theorem can be entered. For this name again it is tested if such a theorem already exists otherwise the theorem is saved with the given name.

5.3 Replacing Subterms

If you want to edit an existing equation you can use the **CHANGE** command. This is useful if you want to make small changes for an existing equation. Otherwise it may be easier to delete this equation and to enter its corrected form.

With the **CHANGE** command one can replace a term or subterm by another term of the same sort. The argument has to be the number of the equation which should be edited. The next step is to select that part of the equation which should be replaced. This is done by moving a "selection pointer" to the desired subterm. TIP prints the selected part of the equation in bold face.

The selection pointer is moved with the characters 's' (sons), 'f' (fathers), 'l' (left) and 'r' (right) through the structure of the equation[1]. Up and down means "go to father" or "go to first son" respectively. Left and right moves the pointer to the left or right sibling argument of the current selection. If you have reached the position in the term which should be replaced you have to use 'a' (accept) or the dot ('.') to tell TIP that this is the substitution position. Now you can enter the new term in the same way as one enters whole equations (see section 5.1) and the selected position will be replaced by this new term.

Let us have a short look at an example. Imagine you typed y instead of z in the associative law and now you want to change this error:

```
TIP> theoremedit demo

   Equations of theorem DEMO:

Equation Nr. 1: add(add(x,y),z) = add(x,add(y,y))
                                                  ^ This is the mistake we want to repair.

Insert (partial) equation. Use '?' for help or '.ABORT' to abort theorem
(2): ?left side? = *
>> .change 1

Substitution Position:
add(add(x,y),z) = add(x,add(y,y))
position> ?
```

1. It is also possible to use 'd' (down), 'u' (up), '<' (left) and '>' (right) instead.

We can use the question mark (‘?’) to see which “commands” are possible here.

```
Possible input here:

 's' or 'd' to go to first son of current term
 'f' or 'u' to go to father of current term
 'r' or '>' to go right to next parameter
 'l' or '<' to go left to previous parameter
 'a' or '.' to accept current position as correct
 '.ABORT' to abort whole substitution
 '?' to get this help


Substitution Position:
add(add(x,y),z) = add(x,add(y,y))
position> r
```

We go “right” to the right hand side of the equation.

```
Substitution Position:
add(add(x,y),z) = add(x,add(y,y))
position> s
```

We go “down” to the first son of the function `add`.

```
Substitution Position:
add(add(x,y),z) = add(x,add(y,y))
position> rsr
```

We go right to the second argument, down to the first son of the inner `add` and then right to the second `y`. As you can see it is possible to give several commands in one line.

```
Substitution Position:
add(add(x,y),z) = add(x,add(y,y))
position> a
```

We tell TIP that we want to substitute this `y`.

```
Selected substitution position:
 add(add(x,y),z) = add(x,add(y,y))

Please give new term for substitution position:
Substitution Term: ?Nat?
>> z
Substitution done. New equation Nr. 1:
(1): add(add(x,y),z) = add(x,add(y,z))

(2): ?left side? = *
>> .exit
```

We corrected the mistake. Now we save the “new” theorem.

```
WARNING: Theorem DEMO already exists!
Should it be overwritten by this new theorem? (y/n) y
Theorem DEMO written.

TIP>
```

It is not possible to replace a function symbol by another one without changing its arguments even if both functions have the same functionality. The reason for this is that TIP considers a function symbol together with its arguments as one term.

5.4 Dealing With Conditional Equations

To keep things simpler the theorem editor of the TIP system can only edit unconditional equations. Conditional equations can be loaded, copied and deleted in the editor, but to change parts of a conditional equation you have to split it into its parts. This is done by the **SPLIT** command. Now the unconditional parts can be treated like any other equation of the theorem. Especially you can use the **CHANGE** command to edit them. Afterwards the changed form of the conditional equation can be build up again. With the **COMPOSE** command several unconditional parts can be connected to a conditional equation.

Moreover, if you want to generate a new conditional equation you have to enter its parts equation by equation and connect them in the end to one conditional equation.

We know that this is not a very smart solution, but up to now there was not enough time to implement a totally new theorem editor. Thus, we adapted the unconditional version for dealing with conditional equations in this way.

6 Special Features of TIP

This chapter contains material about additional features built into TIP. Only the orderings are directly important for proving theorems as described in chapter 4. The other sections describe features which may be interesting for users of the TIP system but which are not necessary for proving theorems with TIP.

6.1 Implemented Orderings

As inductive ordering any monotonic and stable *well-founded ordering* on terms can be used, in particular, orderings that prove the termination of a rewrite system.
There are six different orderings implemented in the TIP system. For every proof of a theorem one of the orderings has to be selected as the current proof ordering. It is not possible to change the proof ordering during a proof session, because this could lead to wrong proofs. Therefore the **SET PARAM ORDERING** command is only allowed in the STANDARD mode.

In version 3.0 the following orderings are available:

- the “normal” subterm ordering
- the lexicographic path ordering
- the multi-set ordering
- the Knuth-Bendix ordering
- the rewrite ordering (only for noetherian specifications)
- the reduce-term ordering (an experimental “semantic” ordering)

The best experiences we had with the lexicographic path ordering. It is powerful enough for most inductive proofs. Therefore the *lexicographic path ordering* is the default in the TIP system. It “includes” the subterm ordering. Thus TIP uses an induction principle, much stronger than the structural induction, which we called ‘*term induction*’ in [Fr89][1].

The rewrite ordering is useful for noetherian specifications and the Knuth-Bendix ordering needs some (tricky) weights for the function symbols. The reduce-term ordering is a variant of the subterm ordering where the user can specify that some function symbols are “reducing” a term.

Of course, there may be situations where the default ordering (lexicographic path ordering) is not powerful enough for your proof. In this case you may experiment with the other orderings.

Subterm Ordering

The subterm ordering $>_{st}$ on terms is defined thus:

$$t >_{st} \mathrm{f}(s_1, \ldots, s_n)$$
$$\Leftrightarrow$$
$$\exists i \in \{1, \ldots, n\}: (t = s_i) \vee (t >_{st} s_i)$$

We recommend not to use this ordering, because it is covered by the lexicographic path ordering. It is still kept available in TIP to enable the emulation of structural induction proofs.

Lexicographic Path Ordering

This ordering is the *default ordering* of the TIP system. It needs a *precedence ordering*, i.e. a well-founded quasi-ordering $\geq_F$ on the set F of the function symbols. (We require F to be finite, so every is well-founded. In TIP this quasi-ordering is derived automatically from the axioms. For function symbols $f, g \in F$ we define $f >_F g$ if g appears at the right-hand side of a rule for *f*, i.e. there is a rule of the form $f(\ldots) \to \ldots g \ldots$. Afterward we build the reflexive and transitive closure of this ordering.

For the lexicographic path ordering $\geq_{lpo}$ we use the version in [De87] which can handle terms with variables and it is a stable ordering. It is defined as:

1. In contrast to our terminology Uday Reddy speaks from “term rewriting induction” [Re90] for a similar prove algorithm.

$$s \geq_{lpo} t$$
$$\Leftrightarrow$$
(1): $t \in X$ and $t \in s$

(2): $s = \mathrm{f}(s_1, \ldots, s_m)$, $t = \mathrm{g}(t_1, \ldots, t_n)$

and

either (2.1): $\mathrm{f} >_F \mathrm{g}$ and $\forall i \in \{1, \ldots, n\}$: $s >_{lpo} t_i$

or (2.2): $\mathrm{f} \sim_F \mathrm{g}$ and $(s_1, \ldots, s_m) \geq_{lpo} (t_1, \ldots, t_n)$

or (2.3): $\exists i \in \{1, \ldots, n\}$: $(s \geq_{lpo} t_i)$

where X in case 1 is the set of variables the lexicographic comparison of tuples in case 2.2 is defined in the following way:

$$(s_1, \ldots, s_m) \geq_{lpo} (t_1, \ldots, t_n)$$
$$\Leftrightarrow$$
(1): $\exists j \in \{1, \ldots, n\}$: $\lfloor (\forall l \in \{1, \ldots, j-1\}: s_l \approx_{lpo} t_l) \wedge (s_j >_{lpo} t_j) \wedge (\forall l \in \{j+1, \ldots, m\}: s >_{lpo} t_l) \rfloor$

(2): $(m \geq n) \wedge (\forall i \in \{1, \ldots, n\}: s_i \approx_{lpo} t_i)$

The larger relation $>_{lpo}$ and the equivalence relation $\approx_{lpo}$ are abbreviations of:

$$s >_{lpo} t \quad \Leftrightarrow_{\mathrm{def}} \quad (s \geq_{lpo} t) \wedge \neg(t \geq_{lpo} s) \qquad \text{and} \qquad s \approx_{lpo} t \quad \Leftrightarrow_{\mathrm{def}} \quad (s \geq_{lpo} t) \wedge (t \geq_{lpo} s)$$

Informally spoken, if the subterm relation is not fulfilled then this ordering tries to sort the terms by comparing their outermost function symbol. If these are equal the arguments are compared in the same way one by one. This way approximately corresponds to the ordering which is used in a dictionary.

Another advantage of this ordering is the possibility to use it for comparing tuples of terms. Comparing two calls of the same function with the lexicographic path ordering corresponds to a lexicographic comparison of both argument tuples.

Case 1 of the above ordering for tuples is the most difficult one, because the sorting of the arguments may cause success or failure. To avoid failure because of wrong argument sorting there is a special command implemented in TIP. By using **SET FUNCTION PERMUTATION <funct_name>** it is possible to rearrange the order of the arguments of this function. This may for example be useful if the first argument of a function grows while the second one gets smaller. In this case the order of the arguments should be permuted to “2, 1".

```
TIP> set function permutation demo_funct
Function DEMO_FUNCT has 2 parameter.
Function DEMO_FUNCT is not permutated.
? Please enter new permutation ?
> 2 1
Permutation of function DEMO_FUNCT:
 2, 1

TIP>
```

The new sorting of the argument numbers has to be entered separated by spaces or by commas. This new order of the arguments is used if this function appears in a check of the smaller-condition for an induction hypothesis.

This command is blocked in the PROVE mode, since changing the argument order during a proof session may lead to unreliable proofs.

The current parameter permutation of a function can be checked by the **SHOW FUNCTION PERMUTATION <funct_name>** command. Only the lexicographic path ordering (and the multi-set ordering which is based on this ordering) uses this information.

Multi-set Ordering

This ordering is useful for comparing (unsorted) sequences. This corresponds to the so-called multi-sets. There are sets which can contain identical elements several times. Sometimes it is meaningful to look upon argument tuples as (unsorted) sequences.

Moreover, the multi-set ordering needs an ordering over the elements of the sequence. In our case this is an ordering over the terms. The multi-set ordering which is implemented in the TIP system uses the lexicographic path ordering for this purpose.

In the multi-set ordering the comparison of the terms $f(s_1, \ldots, s_m) >_{m\,so} g(t_1, \ldots, t_n)$ is transformed to $\{s_1, \ldots, s_m\} \ggg_{ms} \{t_1, \ldots, t_n\}$ id f and g are equivalent in the precedence ordering $\geq_F$.

$$S = \{s_1, \ldots, s_m\} \gg_{ms} T = \{t_1, \ldots, t_n\}$$

$$\Leftrightarrow$$

either $s_i \approx_{lpo} t_j$ and $S\backslash\{s_i\} \gg_{ms} T\backslash\{t_j\}$ for $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$

or $s_i \gtrsim_{lpo} t_{j_1}, t_{j_2}, \ldots, t_{j_k}$ and $S\backslash\{s_i\} \gg_{ms} T\backslash\{t_{j_1}, t_{j_2}, \ldots, t_{j_k}\}$ for $1 \leq j_1 \leq j_2 \leq \ldots \leq j_k \leq n,\ k \geq 0$ and $i \in \{1, \ldots, m\}$

Knuth-Bendix Ordering

This ordering is a quite technical one. It uses an arity function and a function for calculating the frequency of a term. These are defined as follows:

- The arity of a function $f \in \mathrm{F}$ is returned by the function $\mathrm{fct}\colon \mathrm{F} \to \mathbb{N}$.
- The frequency of the occurrences of a term a in a term t is returned by the function $\#\colon W_{(S,F)}(X) \times W_{(S,F)}(X) \to \mathbb{N}$.

For every function symbol and for all variables there has to exist a weight function. Moreover, there are some restrictions for the Knuth-Bendix ordering:

- Let $>_F$ be a partial ordering over F (precedence ordering).
- Let $\mathrm{w}\colon \mathrm{F} \cup \mathrm{X} \to \mathbb{N}$ be a weight function which assigns a natural number to every function symbol $f \in \mathrm{F}$ and every variable $x \in \mathrm{X}$ so that the following conditions are fulfilled:
  - $\forall x \in \mathrm{X}\colon \mathrm{w}(x) = k$ and $k \in \mathbb{N}$ is a constant, i.e. all variables have the same (fixed) weight.
  - $\forall f \in \mathrm{F}\colon (\mathrm{fct}(f) = 0) \Rightarrow (\mathrm{w}(f) \geq k)$, i.e. the weight of a constant function symbol may not be smaller than the weight of a single variable.
  - $\forall f, g \in \mathrm{F}, f \neq g\colon \mathrm{w}(f) = 0 \Rightarrow (f \gtrsim_F g) \wedge (\mathrm{fct}(f) = 1)$, i.e. a function with weight 0 has to have arity 1 and has to be the largest function in the partial ordering $>_F$.
- The weight of a term t is calculated in the following way: $\mathrm{w}(t) = k \sum_{x \in \mathrm{X}} \#(x, t) + \sum_{f \in \mathrm{F}} \#(f, t)\mathrm{w}(f)$, i.e. the weight of every appearance of a function symbol or a variable is summarized to the final weight of the term.

Now we can define the order of two terms in the Knuth-Bendix ordering in the following way:

$$s <_{kb} t$$

$$\Leftrightarrow$$

$\forall x \in \mathrm{X}\colon \#(x, s) \leq \#(x, t)$ and $\mathrm{w}(s) < \mathrm{w}(t)$

or $\mathrm{w}(s) = \mathrm{w}(t)$ and $(s \in \mathrm{X}) \wedge (t \in \mathrm{F})$

or $s = f(s_1, \ldots, s_{\mathrm{fct}(f)}),\ t = g(t_1, \ldots, t_{\mathrm{fct}(g)})$ and $(g >_F f)$

or $s = f(s_1, \ldots, s_{\mathrm{fct}(f)}),\ t = f(t_1, \ldots, t_{\mathrm{fct}(f)})$ with $s_1 = t_1, \ldots, s_{h-1} = t_{h-1}$, $s_h < t_h$ and $1 \leq h \leq \mathrm{fct}(f)$

The weights of function symbols and variables can be defined in the input specification (see section 2.9). Also the necessary precedence ordering can be specified there. The keywords to do this are **kb_weight** and **pr_ordering**. In chapter 2 you can find the details about input specifications for the TIP system.

If no values are given in the input specification then the following default values are used:

$\forall x \in \mathrm{X}\colon \mathrm{w}(x) = 0$, $\forall f \in \mathrm{F}, \mathrm{fct}(f) \geq 1\colon \mathrm{w}(f) = 1, \forall f \in \mathrm{F}, \mathrm{fct}(f) = 0\colon \mathrm{w}(f) = 0$ and $\forall f, g \in \mathrm{F}\colon\ f \nsucc g$.

It is possible to calculate the weights of function symbols and variables by using an implementation of the simplex algorithm, but this is not included in the TIP system.

Rewrite Ordering

This ordering can be used if we have only specifications with terminating axioms. In this case we can use the termination of the rewrite system for defining a correct induction ordering:

$$s <_{\mathrm{rw}} t \quad \Leftrightarrow \quad t \xrightarrow[\Re]{+} s$$

This means that we can apply an induction hypothesis always if at least one rewrite step has been applied to the redex since the generation of this hypothesis. This ordering is normally used for the completion based proofs (see section 6.4).

This ordering can only be used if the union of all “based-on specifications” of the theorem is noetherian. Otherwise it would be possible to prove wrong theorems.

Reduce-term Ordering

This is an experimental ordering and you should only use it if no other ordering is successful! The typical point where the reduce-term ordering may be necessary is a non-constructor function symbol which blocks the smaller condition for applying an induction hypothesis. Moreover this function should reduce its arguments.

This ordering is a kind of mixture of a syntactic and a semantic ordering. Up to now there is no proof for the validity of this ordering for inductive proofs.

With this ordering we try to deal with the problem of functions f which do not enlarge a term. For example reversing a sequence does not make the sequence larger, but a term $\mathrm{rev}(s)$ is always larger than a sequence s if we use a syntactical ordering. Thus, we introduced a “non-enlarging” property for function symbols which specifies one argument of the function that is greater or equal than the whole function call.

> **Definition**: (*non-enlarging*)
>
> A function symbol $f \in \mathrm{F}$ is called *non-enlarging* for its i-th argument if the following holds:
>
> $\exists s\colon\ (\ \mathrm{f}(t_1, \ldots, t_i, \ldots, t_n) \xrightarrow[\Re]{+}\ s\ ) \wedge (\ t_i \geq_{\mathrm{to}} s)$
>
> where $\geq_{\mathrm{to}}$ is an arbitrary termination ordering and $t_1, \ldots, t_n, s$ are ground-terms.

Now we can define the reduce-term ordering $<_{\mathrm{tro}}$ in the following way:

$$s = \mathrm{f}(s_1, \ldots, s_m) <_{\mathrm{rto}} t = \mathrm{g}(t_1, \ldots, t_n)$$

$$\Leftrightarrow$$

either $\exists i\colon s = t_i$

or $\exists i\colon s <_{\mathrm{tro}} t_i$

or f is *non-enlarging* for its i-th argument and $s_i <_{\mathrm{tro}} t$

More informally spoken, the third case means that if the whole term s is “semantically” smaller or equal than one argument s_i and if s_i is “syntactically” smaller than a term t then we conclude that s is smaller than t in our reduce-term ordering. We are using some kind of simultaneous transitivity for a syntactic and a semantic ordering.

To store the information in TIP that a special function symbol is non-enlarging you have to use the **SET FUNCTION REDUCE** command. The parameter of this command is a little bit “tricky”, because you have to give the function symbol *and* the number of the argument. Therefore the argument is the name of a function followed by a period and by the number of the reducing argument, e.g. “rev.1”. If you only use a function symbol as argument than any existing reduce information for this symbol is deleted. To control the reduce information you can use the **SHOW FUNCTION REDUCE <funct_name>** command. This information is only used by the reduce-term ordering!

Of course this way of user defined property may easily lead to wrong proofs if it is not handled very carefully. We recommend to check every application of an induction hypothesis in a proof which was done with the help of the reduce-term ordering. Especially it should be checked if the application is smaller in the semantic ordering one had in mind when specifying the non-enlarging properties.

6.2 Termination and Completeness Check

For every type which is read from a user-specified input file two checks are performed. First it is tested if it is possible to show that the axioms of the specification are terminating and TIP controls if the non-constructor functions are specified completely with relation to the constructor functions. These tests can be suppressed by stating this information in the input file (see section 2.9). Besides, the results of both tests are stored in a so-called property file and if the input file has not been changed until the next TIP session then the test information is taken from the property file instead of performing the checks again. The property file has the same name as the accessory input file with the extension “`.prp`”.

The results of the checks are displayed after parsing the specifications. If the user wants to see this information at a later date (s)he can use the **SHOW PROPERTY <typename>** command. If the **/LONG** option is used then the uncovered cases of the completeness check (if there are some) are displayed too. The **/FILE** option can be used to write this information to a file. The name of this file is the type name together with the extension “`.prp`”.

The Termination Check

A set of rewrite rules R is called *terminating* (or *noetherian*) if there exists no term t which could be rewritten infinitely by axioms from this set. In other words, there is no infinite sequence of the form $t \rightarrow_R t_1 \rightarrow_R t_2 \rightarrow_R \dots$ has to be ruled out for the axioms of R.

One sufficient criterion for the check of the termination is the method of the generalized recursive path ordering [De87]. This check needs a partial ordering on the function symbols (precedence ordering). In TIP we use the lexicographic path ordering and its precedence ordering (see section 6.1, point 2) for the termination check. The lexicographic ordering of the argument tuples may depend on the order of the arguments of a function symbol. For this reason we must test every permutation of the function arguments if necessary.

The check is done in the following way:

For every conditional axiom $s_1{=}t_1$ & … & $s_n{=}t_n \quad \Rightarrow \quad f(p_1, \dots, p_m) \rightarrow r$ we have to test if

- $f(p_1, \dots, p_m) > r$
- $\forall i \in \{1, \dots, n\}$: $f(p_1, \dots, p_m) > s_i$
- $\forall i \in \{1, \dots, n\}$: $f(p_1, \dots, p_m) > t_i$

where > is the lexicographic path ordering.

This means that neither the right hand side nor any of the terms in the conditions is larger than the left hand side. For unconditional equations we only have to check the first point.

If at least one axiom cannot be proved to be terminating then the arguments of its outermost left hand function symbol f are permuted. Of course this permutation has to be applied to all occurrences of f in the axioms and all the axioms have to be tested again. This is done recursively but loops are avoided.

This test is a semi-positive one. This means if the result is “noetherian *`typename`*” then the axioms of this type are proved to be terminating. Otherwise the result of the termination test will be “`noetherian` *`typename`* `except` *`list_of_axioms`*”. Those excepted axioms may either cause non-termination or our implemented test was not powerful enough.

It is well known that termination is undecidable in the general case. Moreover, the implemented termination check may not be powerful enough for difficult termination proofs. Therefore we give the user the possibility to “overrule” the test result of the TIP system. This is done by extending the input specification with the (user-given) information about the termination of a type. The keyword for this information is **noetherian** (see section 2.9).

The information about termination of axioms is used for the rewriting process. Depending on the system parameter **REWRITE_USAGE** non-terminating rewrite rules are applied, not applied or presented to the user. Thus, if the user gives wrong information about the termination of axioms this may cause non-termination of the simplification step during inductive proofs of TIP. More critical is this if the rewrite ordering is used. In this case wrong information about the termination may lead to a wrong proof.

The Completeness Check

A function symbol is called (constructor-) complete if the function can be rewritten for every argument tuple of constructor-ground-terms. The criterion implemented in TIP was taken from [Hu80]. It is a recursive method which does some sort of complete case analysis over the constructor terms for every function symbol. This method also considers the conditions if a function is specified with the help of conditional axioms. The original method was extended to be able to deal with variables occurring several times (non-left linearity).

Axioms which can be processed in the completeness test must have the following form:

$c_1{=}d_1 \wedge \ldots \wedge c_m{=}d_m \Rightarrow f(t_1, \ldots t_n) \rightarrow r$ where $d_1, \ldots, d_m, t_1, \ldots t_n \in W_{(S,C)}(X)$, i.e. constructor terms, $f \in F$ and $c_1, \ldots, c_m, r \in W_{\Sigma}(X)$. Of course, the conditions may be empty.

Technical detail: Every left hand side of a condition of any axiom of the function f forms a “new” parameter position for this function. If this condition appears in an axiom then the right hand side is used as argument for this position, otherwise a new variable is inserted.

The input specifications of the TIP system have a strictly hierarchical structure. Therefore the checks can be done type by type without having to deal with cyclic dependencies. For example, it is not allowed to introduce constructor functions for a sort s in a type T if the sort s has not been declared in T. Therefore we do not have to worry about constructors declared later when TIP checks the completeness of a type.

The results of the completeness test are not used for the inductive theorem proving. They just signal the user that there may have some errors in his/her specification.

The test algorithm is also used for checking the completeness of a case analysis during the proof of a theorem.

6.3 Unification and Matching in the AC Theory

The unification of two terms t_1 and t_2 means to find a substitution σ (called *unifier*) such that both terms are syntactically equal when this substitution is applied to them: $t_1\sigma = t_2\sigma$. Normally there exist several unifiers and therefore one is interested in the *most general unifier* for two terms. A well known (exponential) algorithm from J.A. Robinson (1969) solves this problem. Unification is used for narrowing steps. In the meantime, there are more efficient unification algorithms available.

Matching one term with another is something quite similar. Here only the variables of one term may be substituted, the other term is fixed. Substitutions which fullfil this task are called matcher: $t_1 = t_2\sigma$. Again we are mainly interested in the most general matcher. Matching algorithms normally can be developed with small modifications from unification algorithms. Matching is used for rewriting and rule applications.

Adding axioms like the commutativity to a term rewrite system is difficult because such an axiom cannot be directed. Putting the commutativity of f as equation $f(x, y) = f(y, x)$ in the scheme of directed rewriting rules will always cause non-termination of the rewrite system. Therefore one unifies or matches respectively *modulo commutativity*.

More generally one can do unification and matching modulo of a set of equations E. Then we speak of *E-unification* and *E-matching*. These procedures allow us to handle the (normally non-orientable) equations in E. The most frequently used sets of equations are the associative and commutative theories, *AC* for short.

In 1975 M. Stickel introduced the first correct and complete AC-unification algorithm. But it took nearly 10 years until F. Fages could prove that this algorithm is really terminating. The central idea of AC-unification is the transformation of the problem to the solving of linear diophantine equation. Another problem for AC-unification and AC-matching is that in most cases there exist several most general unifiers or matchers respectively. AC-unification is NP-complete!

In version 3.0 of the TIP system an AC-unification and an AC-matching algorithm is included. It was implemented by G. Bittmann [Bi92]. These algorithms are used for inductive theorem proving and for completion based theorem proving (see section 6.4). Besides, there are two commands available in TIP to find the AC-unifiers and the AC-Matchers for two given terms. To avoid new data structures in TIP both commands deal with theorems. The left hand and the right hand side of the conclusion of the first (unconditional) equation of the theorem are used as the two terms which should be unified or matched respectively. The commands to call the algorithms are **UNIFY <theo_name>** and **MATCH <theo_name>**. If you match the two terms then the first one (i.e. the left hand side) is viewed as that term which can be substituted by the matcher.

Of course the TIP system has to know which function symbols are associative and commutative. This is specified in the input files after the keyword **ac_info** (see section 2.9). To check which functions of a type have these properties you can use the **SHOW PROPERTY <typename>** command. The **SET FUNCTION AC <func-name>** command can be used to define the AC property for a special function. If the **/FALSE** option is used the AC property deleted.

Example: Assume we have a theorem named AC_DEMO which consists of the following equation:

```
add(z,succ(w)) = add(add(x,y),succ(x))
```

Here `w`, `x`, `y` and `z` are variables and the function `add` is known to be AC. By way of contrast `succ` is not AC. (Only functions with two arguments and the same argument and result sort can be AC!)

```
TIP> show property nat
noetherian NAT
complete NAT
ac_info NAT add, mult

TIP> unify ac_demo
                 AC-unifiers of

 add(z,succ(w)) = add(add(x,y),succ(x))

w<-x, z<-add(y,x)
z<-add(add(succ(x),y),X102), x<-add(succ(w),X102)
z<-add(succ(x),y), x<-succ(w)
z<-add(succ(x),x), y<-succ(w)
z<-add(add(succ(x),X101),x), y<-add(succ(w),X101)

TIP> match ac_demo
              AC-matchers of

add(z,succ(w)) = add(add(x,y),succ(x))

w<-x, z<-add(y,x)

TIP>
```

Both algorithms are able to handle both AC functions and non-AC functions. Moreover, any combination of both classes can be handled[1]. By default they only show a list of the most general unifiers or matchers, but there exists a parameter in the algorithms to list all found substitutions, too.

Another important algorithm which is used in TIP is *AC-equal*. As you would expect, this algorithm checks if two terms are equal modulo AC. No substitution is calculated here. The result of this check is just a true or false statement. If this check shows that a goal is “trivial modulo AC” then in the proof listing the following statement appears:

1. Some algorithms described in the literature cannot handle AC functions which are arguments of non-AC functions.

PROOF OF THEOREM

```
add(add(X1,X2),X3) = add(X3,add(X2,X1))
add(add(X1,X2),X3) = add(add(X1,X2),X3)                         (by AC-equal)
```

THEOREM PROVED

In the TIP standard type the functions `and` and `or` of `BOOL` and `add` and `mult` of `NAT` are declared to have the AC property.

6.4 Completion Based Theorem Proving

In 1970 D.E. Knuth and P. Bendix introduced a procedure which could decide if two terms are equal w.r.t a set of axioms. This was based on an algorithm which generates a canonical[1] rewrite system out of a terminating[2] but not confluent rewrite system (the *completion algorithm*). In the course of time this procedure was generalized in such a way that it could do inductive proofs. In the literature there are different names for this kind of inductive theorem proving: *inductionless induction* or *completion based theorem proving* or *proving by consistency.*

The central idea of this proof procedure is to have a confluent term rewrite system, add a theorem and then to check if the rewrite system is still conflict free. Both, the transformation of a rewrite system in a confluent one and checking if a "extended" rewrite system is conflict free can be done with the completion algorithm of Knuth and Bendix. Therefore it is also checked if the rewrite system is ground-reducible.

The theorem which should be proved is added as a new (directed) rewrite rule in the underlying term rewrite system. This underlying rewrite system has to be confluent. Then all the *critical pairs*[3] between the theorem and the rewrite system are calculated and simplified if necessary. The new rules which result from this are treated in the same way. This is continued until a conflict appears or no more new critical pairs can be generated. If a conflict appears then the theorem is disproved. In the second case the whole rewrite system is confluent and the theorem is proved.

Of course adding a theorem to a rewrite system means that the theorem has to be directed. To deal also with undirectable rules and lemmas this kind of theorem provers normally are strongly based on unification modulo a set of equations (see section 6.3).

It would lead too far to describe the whole theory of completion based theorem proving in this manual. The interested reader is referred to the literature, e.g. [KM87] and [Du91].

In TIP the completion algorithm is started with the **COMPLETE** command. The parameter of this command may either be a type or a theorem. In both cases the specifications (including all "based on" specifications) are transformed in one large ("flat") specification. This specification now is completed. If the argument was a theorem name and the completion of the specification was successful then the theorem is added to the specification now and the completion algorithm is called again.

The completion algorithm implemented in TIP is not a very optimized one. It should be used only for small examples because otherwise it may run out of memory very quickly. For example the standard type `NAT` cannot be checked for confluence with this completion algorithm. If you want to work with this command you should keep your specification small. Remove every declaration which it not necessary for proving your theorem.

Inductionless induction proofs are very much designed for automatic proving. There is nearly no interaction with the user except for directing difficult equations. The proof output is a confluent rewrite system and the proof listing is very hard to understand for non-experts.

Example: We want to prove that reverting any sequence twice will leave the sequence unchanged.

---

1. A rewrite system is called canonical if it is noetherian and confluent.
2. noetherian
3. A critical pair can be generated if the left hand side of a rule and a subterm of the left hand side of a rule can be unified. In this case there exists a term which can be rewritten with two rewrite rules in different new terms.

Therefore we need a small specification for sequences and reverting sequences. The constructor function for a sequence are `empty` and `append( )`. The function `conc` concatenates two sequences and the function `rev` reverts a sequence. The theorem REVREV consists of the equation:

```
rev(rev(s)) = s
```

The completion of the type REV shows that it is confluent. Then the equation from the theorem is added as a new rule. Again the completion algorithm terminates successfully. This proves that the theorem is correct for the specification REV. Finally a new specification REVREVC is generated. This type is a complete version of the theorem REVREV and all its "based on" specifications.

```
TIP> complete REVREV
********************************************************************************
*                    COMPLETION OF THE TYPES OF THEOREM REVREV                *
********************************************************************************

                         INSERT NEW RULE IN TRS

 R1 rev(empty) -> empty
GENERATED FROM AN AXIOM OF THE SPECIFICATION
================================================================================
…
                         INSERT NEW RULE IN TRS

 R4 conc(append(X13,X14),X15) -> append(X13,conc(X14,X15))

 GENERATED FROM AN AXIOM OF THE SPECIFICATION
================================================================================
 SUCCESSFULL COMPLETION OF THE TYPES OF THEOREM REVREV

 ACTUAL TRS:

 R1 rev(empty) -> empty
 R2 conc(empty,X2) -> X2
 R3 rev(append(X6,X7)) -> conc(rev(X7),append(X6,empty))
 R4 conc(append(X13,X14),X15) -> append(X13,conc(X14,X15))

 ACTUAL TRS CONTAINS 4 RULES
================================================================================

********************************************************************************
*                             START PROOF OF THEOREM REVREV                   *
********************************************************************************

                         INSERT NEW RULE IN TRS

 R5 rev(rev(X20)) -> X20

 GENERATED FROM A THEOREM
================================================================================
                         INSERT NEW RULE IN TRS

 R6 rev(conc(rev(X23),append(X22,empty))) -> append(X22,X23)

 GENERATED FROM A CRITICAL PAIR OF R5 AND R3
================================================================================
                         INSERT NEW RULE IN TRS

 R7 rev(conc(X29,append(X28,empty))) -> append(X28,rev(X29))

 GENERATED FROM A CRITICAL PAIR OF R6 AND R5
================================================================================
                         DELETE RULE FROM TRS

 R6 DELETED BY R7
 LEFT SIDE REDUCED USING: R7 R5
================================================================================
 SUCCESSFULL PROOF OF THEOREM REVREV

 ACTUAL TRS:

 R1 rev(empty) -> empty
 R2 conc(empty,X2) -> X2
 R3 rev(append(X6,X7)) -> conc(rev(X7),append(X6,empty))
 R4 conc(append(X13,X14),X15) -> append(X13,conc(X14,X15))
 R5 rev(rev(X20)) -> X20
```

```
R7 rev(conc(X29,append(X28,empty))) -> append(X28,rev(X29))

ACTUAL TRS CONTAINS 6 RULES
================================================================================
NEW TYPE REVREVC GENERATED
================================================================================
TIP>
```

The completion algorithm implemented in TIP uses the AC-unification and AC-matching as described in section 6.3. Therefore it is useful to check the AC definitions of the used types. The only proof listing that is available for completion based proofs is the output during the completion procedure and the new complete type which is generated.

This completion algorithm cannot handle conditional axioms or conditional theorems. If conditional axioms appear in the types or a conditional equation in the theorem then they are simply ignored. Most of the implementations of the Knuth-Bendix algorithm cannot handle conditions.Therefore this behavior should not be too unusual.

7 Future Developments

Of course there are many points left for improvements of the current version of the TIP System. Some of them are planed for the next time and others will be postponed to a later date until some student or someone else is found who is interested in doing these implementations. The following list shall give you an impression of our ideas:

- Constructing a user-interface for the X-window-system. This should include a “real” graphical tree display and the possibility to control a proof with the help of the mouse. Commands could be selected from menus and the system parameter could be selected in a separate window. Maybe this will be done till 1994 from a student as his diploma thesis.
- Finding some good heuristics for enriching the conditions with new equations. This means we have to look at conditional axioms which could be applied but of which the conditions cannot be evaluated to true. These conditions should be saved in a list and if one term appears more than one time then it could be a promising candidate for building a new goal condition together with a new variable. We hope that this will be implemented till 1994.
- More orderings could be implemented or the user should get the possibility to enter an induction ordering together with the specification (like it is done in the Boyer-Moore prover).
- A better way of displaying large terms would be necessary. Maybe this can be for a future X-Version. Then it would be possible to use the Σ sign instead of the string `sum(…)`. Also a better support of mixfix notation would be useful. Or at least a better form of pretty printing for larger terms.
- A better connection of “natural” induction and completion based induction (see section 6.4) would be a promising step in the direction of higher automatization of inductive proofs.
- The proof algorithm could be made more flexible. This means the order of the proof steps should not be fixed. If the user is able to design special proof algorithms and to load these into the TIP system then one could use different strategies to solve different sorts of proof.
- A protocol of every proof session should be stored in a file. This protocol should contain all user commands which are important for the proof session, i.e. no show commands. Also it should be possible to use this protocol as input for the TIP system. Thus, you could rerun your proof session if you want. Especially it would be possible to continue an incomplete proof by using the protocol to the same state at which one “left” the prover before. This protocol together with the specification and the final proof listing would be a complete storage of a proof. Maybe this will be done till 1994 by some students.
- Implementing a specialized algorithm for checking the confluence of an input specification would be useful. Up to now the implemented completion procedure can be used for this purpose.
- Testing the power of the proof algorithm of the TIP system with many non-trivial proofs. Especially several comparisons with other inductive prove systems concerning power, speed, quality of the detected proofs and user interaction are necessary to classify and improve TIP.
- Adding a proof checker of another development to check the proof listings generated by TIP. The TIP system is not verified and thus, we can not rule out that there are still some errors inside. Using two totally different systems to generate and to control a proof would bring the reliability of TIP to a standard which is usual for the request of real-world verification.
- A component for finding and suggesting possibly necessary lemmas automatically would be a very great step for inductive theorem proving. This means that parts of the proof tree are analyzed to find out why the induction hypothesis - which is the most important step in inductive proofs - can not be applied. Then a reformulation of the theorem or a possible lemma should be derived from this analysis. This research is directed to knowledge based systems, because it is obvious that there is no standard solution for this process.

8 References

BB92] K.H. Bläsius, H.-J. Bürckert (eds.): Deduktionssysteme - Automatisierung des logischen Denkens. Oldenbourg, München, (1992).

[Bi92] G. Bittmann: *AC-Unification für algebraische Spezifikationen*, diploma thesis at the University of Passau, 1992.

[BM79] R.S. Boyer, J.S. Moore: *A Computational Logic*, Academic Press, 1979.

[BM88] R.S. Boyer, J.S. Moore: A Computational Logic Handbook. Academic Press, (1988).

[De82] N. Dershowitz: *Orderings for term rewriting systems*, Journal of Theoretical Computer Science 17(3), 1982, pp. 279-301.

[De87] N. Dershowitz: *Termination of Rewriting*, Journal of Symbolic Computation, Vol. 3, 1987.

[Du91] D. Duffy: *Principles of Automated Theorem Proving*, Wiley, 1991.

[FH92] U. Fraus, H. Hußmann: *A Narrowing-Based Theorem Prover*, Proceedings of the IMA Conference, July 1990, University of Stirling, Oxford University Press, 1992.

[FK92] M. Frañová, Y. Kordratoff: Simplifying Implications in Inductive Theorem Proving: Why and How? Rapport de Recherche no. 788, Universite de Paris-Sud, Orsay, (1992).

[Fr85] L. Fribourg: *Handling function definitions through innermost superposition and rewriting*, Proceedings RTA 85 Conference, LNCS 202, pp. 325-344.

[Fr89] U. Fraus: *Eine interaktive Umgebung für Induktionsbeweise über algebraischen Strukturen*, diploma thesis 1988, report MIP-intern 04/89, Universität Passau, 1989.

[Fr92] U. Fraus: A Calculus for Conditional Inductive Theorem Proving. In Proceedings of the CTRS'92“, Nancy, (Eds.: M. Rusinowitch, J.L. Remy) LNCS 656, Springer Verlag, (1993).

[Ge86] A. Geser: *A specification of the INTEL 8085 microprocessor: A case study*, Report MIP-8608 Universität Passau, 1986.

[GH85] A. Geser, H. Hußmann: *Rapid prototyping for algebraic specifications - examples for the use of the RAP system*, Report MIP 8517 Universität Passau, 1985.

[GH86] A. Geser, H. Hußmann: *Experiences with the RAP system - a specification interpreter combining term rewriting and resolution*, Proceedings ESOP 86 Conf., LNCS 213, pp. 339-350.

[GL91] B. Gramlich, W. Lindner: *A Guide to UNICOM, An Inductive Theorem Prover Based on Rewriting and Completion Techniques*, SEKI Report SR-91-17 (SFB), Universität Kaiserslautern, 1991.

[Hu80] G. Huet: *Confluent Reductions: Abstract properties and applications to term rewriting systems*, Journal of the ACM 27(4), 1980, pp. 797-821.

[HH82] G. Huet, J.-M. Hullot: *Proofs by Induction in Equational Theories With Constructors*, Journal of Computers and System Science Vol. 25, 1982, pp. 239-266.

[HO80] G. Huet, D. C. Oppen: *Equations and rewrite rules, a survey*, In: R. V. Book (ed.): Formal language theory - perspectives and open problems, Academic Press 1980.

[HR87] H. Hußmann, C. Rank: *Specification and prototyping of a compiler for a small applicative language*, in the Proc. 1st ESPRIT METEOR workshop „Algebraic methods: Theory, tools, and applications“, Universität Passau, June 9-11, 1987

[Hu85] H. Hußmann: *Unification in conditional-equational theories*, Report MIP-8502 Universität Passau, 1985.
Short version in: Proceedings EUROCAL 85 Conference, LNCS 204, pp. 543-553.

[Hu92a] Heinrich Hußmann: A case study towards algebraic verification of code generation. In: Nivat, Rattray, Rus, Scollo (eds.): Algebraic Methodology and Software Technology (AMAST '91), Workshops in Computing, Springer Verlag, (1992).

[Hu92b] D. Hutter: Vollständige Induktion. In Deduktionssysteme - Automatisierung des logischen

Denkens (Eds.: K.H. Bläsius, H.-J. Bürckert), Oldenbourg, München, (1992).

[KB70] D.E. Knuth, P.B. Bendix: Simple Word Problems in Universal Algebras. Computational Problems in Abstract Algebras (Ed.: J. Leech). Pergamon Press, (1970), pp.263-297.

[KM87] D. Kapur, D. Musser: *Proof by Consistency*, Artificial Intelligence 31, 1987, pp. 125-157.

[Pa87] L.C. Paulson: Logic and Computation. Cambridge University Press, (1987).

[Pa88] P. Padawitz: Computing in Horn Clause Theories. Springer, (1988).

[Re90] U. S. Reddy: *Term Rewriting Induction*, CADE 10, LNCS 249, Springer, 1990, pp. 162-177.

[Re85] P. Rety, C. Kirchner, H. Kirchner, P. Lescanne: *NARROWER: a new algorithm for unification and its application to logic programming*, Proceedings RTA 85 Conference, LNCS 202, pp. 141-155.

[Ri92] K. Rieger: Ein vervollständigungs-basierter Theorembeweiser für algebraische Spezifikationen. Diplomarbeit, Universität Passau, (1992).

[Wa92] C. Walther: Computing Induction Axioms. In Proceedings of the LPAR'92 (Ed.: A. Voronkov), LNCS 624, Springer Verlag, (1992)

[Wi90] M. Wirsing: *Algebraic Specification*. In: J. van Leeuwen (Ed.): *The Handbook of Theoretical Computer Science*, Volume B, Amsterdam, Elsevier (1990), pp. 675-788.

[Wi91] M. Wirsing: Proofs In Structured Specifications. MIP-9008, Universität Passau, (1991).

[Zh92] H. Zhang: Implementing Contextual Rewriting. In Proceedings of the CTRS'92, Nancy, (Eds.: M. Rusinowitch, J.L. Remy) LNCS 656, Springer Verlag, (1993).

Appendix

Appendix A: Glossary

AC unification . . . . Doing **unification** where some function symbols are treated as ***asso**ciative* and ***c**ommutative*.

(Constructor-)Complete . . . . A function symbol f is called **constructor-complete** (w.r.t. a set C of constructor functions) if *every term* of the form $f(t_1, \ldots, t_n)$, such that each t_i is a constructor ground term, admits a rewrite step at the top position.

Context. . . . . A term with a 'hole' in it. A context models the part of a term *above* a given position (cf. subterm).

Contradiction, Clash . . . . . . A goal **clashes**, i.e. it forms a **contradiction**, if it contains an equation of the form $f(t_1, \ldots, t_n) = g(u_1, \ldots, u_m)$ where f and g are two different "*irreducible function symbols*".

Decomposition . . . . The *replacement* of the equation $f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)$ by the system $t_1 = u_1, \ldots, t_n = u_n$ is called **decomposition**. This step is only correct if f is a "*irreducible function symbols*".

Diophantine equation . . . . . An equation of the form $\sum_{i=1}^{m} a_i x_i = \sum_{j=1}^{n} b_i y_i + c$ where $a_1, \ldots, a_m$, $b_1, \ldots, b_n$, c, m and n are positive integers is called linear **diophantine equation**. If $c = 0$ then the equation is called homogeneous, otherwise inhomogeneous. One is interested only in (positive) integer solutions for the vectors x and y. Such equations are useful for *AC unification*.

Directable equation . . . . . . An equation $l = r$ (or $r = l$) is **directable** (or **orientable**) w.r.t. a given well-founded ordering if $l > r$ holds. In this case the equation can be replaced by a rewrite rule $l \to r$. Important for completion based theorem proving.

Failed Goal. . . . . A goal **fails** if *no prove step can be applied* to this goal and if it is neither solved nor disproved. This means we have a dead end in our proof tree and we do not know anything about its state. Normally such cases indicate that the proof system is not powerful enough for this proof.

FTP. . . . . This is an abbreviation for ***F**ile **T**ransfer **P**rotocol*. The program `ftp` allows a user to transfer files to and from a remote network site. This means, if you are connected to some non-local network, then you can get programs and documents form other computers.

Generalization . . . . A **generalization** of an equation means that all occurrences of a *term* in this equation are *replaced by a new variable*. A generalization is called inhomogeneous if not all occurrences of a term are replaced by a new variable,

Ground term . . . . A term *without variables*. Such a term is build up only from function symbols (including constants).

Goal . . . . A *system of* (conditional) *equations* which should be solved during a proof. Besides, a goal contains *information* necessary for the proof algorithm. Every goal is a node in the proof tree. Goals are subtasks of the proof obligation.

Irreducible function symbol . . A function symbol f is called **irreducible** if there *exists no axiom for it*, i.e. no rewrite rule of the form $f(\ldots) \to r$ or $C \Rightarrow f(\ldots) \to r$ exists. Therefore such a function can never be evaluated. Decomposition and contradictions need irreducible function symbols.

Lemma . . . . Is a synonym to *auxiliary theorem*.

More special . . . . A term t_2 is called **more special** than a term t_1 if there exists a substitution σ such that: $t_1\sigma = t_2$

Redex. . . . . A **redex** (**red**ucible **ex**pression) (pl. *redices*) is a *(sub)term* in a (conditional) equation that matches the left-hand side of a rule. One can view a redex as *a subterm where a rule can be applied*.

Signature . . . . A **signature** is a tuple $\Sigma = (S, F)$, where S is a *set of sort symbols* and F is a *set of function symbols*. To every function symbol $f \in F$ are assigned a finite sequence of sort symbols (its argument sorts) and a sort symbol (its result sort).

Specification . . . . A **specification** $T = (\Sigma, A)$ is a tuple, where Σ is a *signature* and A is *set of* (conditional) *axioms*.

Sort . . . . The name of a *set of objects* in TIP.

Subcommand. . . . . A *part of a longer command* which appears not in the first place.

Subterm . . . . A **subterm** of a term is either the term itself or a proper subterm, i.e a subterm of one of its arguments. A subterm is the part of the term *below* a given position (cf. context).

Term . . . . A **term** is either a *variable* or an *expression* of the form $f(t_1, \ldots, t_n)$ where f is a function symbol and the t_i are terms of the right sort.

Theorem . . . . A (system of) *conditional equation(s)* where all occurring variables are *universally quantified* on the outermost level. A **theorem** is also the input text of a proof statement for the TIP system.

Twin step. . . . . A *combined proof step* which consists of the inflation of a variable by applying a lemma or a hypothesis followed by a generalization step.

Type . . . . A **type** is the *input text of a specification* for the TIP system. (Not be confused with the notation of *sort*.)

Appendix B: Installation of the TIP System

The TIP system is available via anonymous FTP from the server of Bavarian Research Center for Knowledge Based Systems (FORWISS) *forwiss.uni-passau.de*. The directory is `pub/local/tip`.

The theorem prover comes as compressed tar-file `tip.tar.Z`, including source code, a binary file usable on SUN4–systems, examples, and documentation.

Instructions for obtaining this file:

1. Connect to the FORWISS ftp server:

```
ftp forwiss.uni-passau.de
```

2. Login as "anonymous" and use your user-id as password. Then go to the ftp directory:

```
cd pub/local/tip
```

3. Change the mode to binary by typing

```
binary
```

4. Then to transfer the compressed tar file type:

```
get tip.tar.Z
```

5. Then type

```
quit
```

to exit.

We assume that you have copied this file to your home directory. Unpacking it with the command

```
zcat tip.tar.Z | tar xf -
```

creates the following directories and files:

- `tip/bin` SUN4-binary, TIP-standard-files and shell-script
- `tip/src` Pascal- and C-sources with a Makefile
- `tip/docs` some documentation, last changes and a manual page for the *man* command
- `tip/examples` several examples of types and theorems
- `README` a short description of the TIP system and its installation
- `COPYRIGHT` the GNU copyright text

To compile the TIP source code you need a Pascal compiler which supports a module concept and a C compiler which supports ANSI–C. The command for the Pascal compiler is assumed to be `pc` and the command for the C compiler `gcc`. You can change these default settings by adapting the makefile variables `P_COMPILER` and `C_COMPILER`, respectively.

To begin with, we will describe a private installation procedure for playing with TIP:

If your system can execute SUN4 code, after the following procedure, assuming that you have copied `tip.tar.Z` to your home directory, you can use the TIP system:

(1)	`zcat tip.tar.Z | tar xf -`	creating the directory `tip`
(2)	`setenv PATH=$PATH:$HOME/tip/bin`	here is the executing shell script
(3)	`setenv TIPBIN=$HOME/tip/bin`	here is the binary
(4)	`setenv TIPSTD=$HOME/tip/bin`	and here are the system files

If your system isn't SUN4–code-compatible or if you want to recompile TIP anyway, you may take steps according to the following commands, again assuming you have copied `tip.tar` to your home directory:

(1)	`zcat tip.tar.Z | tar xf -`	creating the directory `tip`
(2)	`cd tip/src`	
(3)	`make clear`	deleting the old binary
(4)	`make`	creating the binary `tipprg`

(5) `setenv PATH=$PATH:$HOME/tip/bin` here is the executing shell script

(6) `setenv TIPBIN=$HOME/tip/bin` here is the binary

(7) `setenv TIPSTD=$HOME/tip/bin` and here are the system files

These last two environment settings ((2)–(4) and (5)–(7) respectively) should be made permanent by including them in the start-up-sequence.

Now you can start the TIP system with the command `TIP`.

For the purpose of opening TIP to the public change the paths in `~/tip/src/Makefile`, in which directories you want to place the TIP–shell (`TIPSH`), TIP–binary (`TIPBIN`), the TIP–standard-files (`TIPSTD`) and the manual page (`TIPMAN`), according to the file structure on your system. Then execute

(1) `cd tip/src`

(2) `make clear`

(3) `make`

(4) `make install`

(5) `setenv PATH=$PATH:`<your TIP–shell path as in the Makefile>

(6) `setenv TIPBIN=`<your TIP–binary path as in the Makefile>

(7) `setenv TIPSTD=`<your TIP–standard–files path as in the Makefile>

Again you can skip the steps (2) and (3) if your system is SUN4 code-compatible.

Appendix C: Start-up Files

Five files which are necessary to start the TIP system successfully:

- `tipprg` The executable TIP code.
- `tipstd` The standard types BOOL and NAT.
- `tiphelp` The help text for the HELP command.
- `tiperror` The text which is displayed for error messages.
- `TIP` This shell script does all the necessary links and preparations before starting the prover itself. The name of this script is the command which starts TIP for your local platform. It should be either in any directory of the UNIX command path (see '`echo $PATH`') or in the same directory as your specifications.

Two other important files should be mentioned here: The file `tipinit` may contain TIP commands which should be executed automatically after every start-up of the TIP system. If this file is not existing in the current directory then - if available - a link to `$HOME/.tiprc` is set. Otherwise the empty file is used. These start-up commands (see appendix 3.5.2) can be used to load special types and theorems or to set some kind of personal environment.

The file `tipfiles` is created after starting the TIP system. It is containing a list of all files which have to be read into the TIP system. These are all UNIX files which have been stated on the command line. This way of handling the command parameters is necessary for dealing with several input files simultaneously. In addition this list of input files is needed when all specifications should be reread by using the **READ** command.

The use of this shell script and the linking of files to current directory is necessary because the executable '`tipprg`' is programed in standard PASCAL and therefore it is neither possible to read the UNIX command line nor can environment variables like '`$TIPSTD`' be used which are depending on your local installation. Thus, the shell script can be seen as some kind of frame which prepares all the necessary information for the theorem prover.

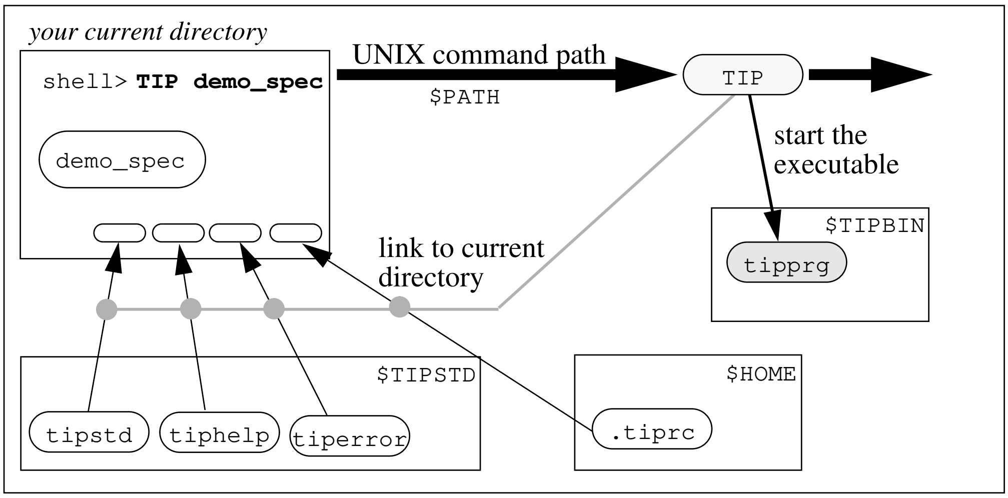


Fig. 7: Events when execution the shell script 'TIP'.

This means, when you execute the shell script '`TIP`'[1] then the following actions are carried out:

- if the files '`tipstd`', '`tiphelp`' and '`tiperror`' do not exist in the current directory then link these files from the place which is stored in the environment variable '`$TIPSTD`' to the current directory;
- list all file names of the UNIX command line in the file '`tipfiles`';
- interpret flags which have been used in the UNIX command line;
- link the PASCAL program '`tipprg`' from '`$TIPBIN`' to the current directory and start it;
- remove all links and the file '`tipfiles`' after the '`tipprg`' has been finished.

1. It is possible to rename this file to any other name that should be used as command for starting the TIP system. But you should be careful if you want to use the name 'tip', because normally it already exists as command for UNIX systems.

Appendix D: Standard Type Definitions

```
type BOOL
{The Boolean values}
     sort Bool

     cons true,false : Bool
     func not : (Bool)Bool,
          and : (Bool,Bool)Bool,
          or  : (Bool,Bool)Bool,
          impl: (Bool,Bool)Bool, {implication }
          equiv : (Bool,Bool)Bool {equivalence }

  axioms all (x : Bool)

     not(true) -> false,
     not(false) -> true,

     and(true,x) -> x,
     and(x,true) -> x,
     and(false,x) -> false,
     and(x,false) -> false,

     or(false,x) -> x,
     or(x,false) -> x,
     or(true,x) -> true,
     or(x,true) -> true,

     impl(x,x) -> true,
     impl(true,false) -> false,
     impl(false,x) -> true,
     impl(x,true) -> true,

     equiv(x,x) -> true,
     equiv(true,x) -> x,
     equiv(x,true) -> x
endoftype

noetherian BOOL
complete BOOL
ac_info BOOL and,or

display BOOL
 not = "~ _",
 and = "_ ^ _",
 or = "_ V _",
 impl = "_=)_",
 equiv = "_(=)_"
```

```
type NAT
{The natural numbers }

    basedon BOOL
    sort    Nat
    cons    zero : Nat,
            succ : (Nat)Nat       { successor }
    func    pred : (Nat)Nat,      { predecessor }
            add : (Nat,Nat)Nat,   { addition }
            sub : (Nat,Nat)Nat,   { subtraction }
            mult : (Nat,Nat)Nat,  { multiplication }
            div  : (Nat,Nat)Nat,  { division }
            mod  : (Nat,Nat)Nat,  { modulo }
            equal_Nat : (Nat,Nat)Bool, { equality }
            lt : (Nat,Nat)Bool,   { less than }
            gt : (Nat,Nat)Bool,   { greater than }
            le : (Nat,Nat)Bool,   { less equal }
            ge : (Nat,Nat)Bool    { greater equal }

    axioms all (x,y,z,r : Nat)

      pred(zero) -> zero,
      pred(succ(x)) -> x,
      add(x,zero) -> x,
      add(x,succ(y)) -> succ(add(x,y)),
      add(zero,x) -> x,
      add(succ(x),y) -> succ(add(x,y)),
         sub(zero,x) -> zero,
      sub(succ(x),zero) -> succ(x),
      sub(succ(x),succ(y)) -> sub(x,y),
      mult(x,zero) -> zero,
      mult(x,succ(y)) -> add(mult(x,y),x),
      mult(zero,x) -> zero,
      mult(succ(x),y) -> add(mult(x,y),y),
      lt(x,y) = true  =>  div(x,y) -> zero,
      lt(x,y) = false =>  div(x,y) -> succ(div(sub(x,y),y)),
      lt(x,y) = true =>  mod(x,y) -> x,
      lt(x,y) = false =>  mod(x,y) -> mod(sub(x,y),y),

      le(x,x) -> true,
      le(0,x) -> true,
      le(succ(x),0) -> false,
      le(succ(x),succ(y)) -> le(x,y),

      ge(x,y) -> le(y,x),
      gt(x,y) -> not(le(x,y)),
      lt(x,y) -> not(ge(x,y)),

      equal_Nat(x,x) -> true,
      equal_Nat(0,succ(x)) -> false,
      equal_Nat(succ(x),0) -> false,
      equal_Nat(succ(x),succ(y)) -> equal_Nat(x,y)
endoftype

noetherian NAT
complete NAT
ac_info NAT add,mult

display NAT
  add = "_+_",
  mult = "_*_",
  equal_Nat = "_==_",
  le = "_<=_",
  lt = "_<_",
  ge = "_>=_",
  gt = "_>_"
```

Appendix E: Larger Examples

Appendix E.1: Double Reversion of a Sequence

This example shows the application of a twin-step. In the proof listing below this step is represented by the application of a hypothesis directly followed by a generalization. The input file contains a specification for sequences which contains a function `conc` for the concatenation of two sequences and a function `rev` for reversing a sequence. The theorem `REVREV` of double reversion is one of the standard examples for completion based induction.

Please notice that we use `append` as a constructor function and `conc` for the concatenation. Some other specification languages use `append` for the concatenation, and `cons` for the constructor instead.

```
type SEQ
  basedon NAT,BOOL
  sort Seq
  cons empty: Seq,           { empty sequence }
       append: (Nat,Seq)Seq  { append one number }
  func conc: (Seq,Seq)Seq,   { concatenate }
       rev: (Seq)Seq

axioms all (s, s1, s2: Seq, n: Nat)
  {1} conc(empty,s1) -> s1,
  {2} conc(append(n,s1),s2) -> append(n,conc(s1,s2)),
  {3} rev(empty) -> empty,
  {4} rev(append(n,s)) -> conc(rev(s),append(n,empty))
endoftype


theorem REVREV
  basedon SEQ,NAT

  forall x: Seq
goals
  rev(rev(x)) = x
endoftheorem
```

The central problem for this proof is, that there is no axiom for evaluating a term of the form

`rev(conc(…,…))`

Therefore we need to do a case analysis for the arguments of `conc`. But the first argument is blocked by a `rev(X…)` term. And exactly this troublesome term is “moved” to the right hand side by a twin-step and

`rev(conc(X4,append(X2,empty))) = append(X2,rev(X4))`

is “found” as a necessary lemma which is immediately proved with an “independent” induction proof. Version 3.0 of the TIP system can prove the theorem `REVREV` automatically and supplies the following proof listing:

```
TIP System Version 3.0 SUN 4.1/UNIX                           Listing_of_Proof


PROOF OF THEOREM

 rev(rev(X1)) = X1

 COMPLETE CASE ANALYSIS on X1                                          => [1]
 (Generated from the rules for function rev)

1. CASE: X1 = empty

   - rev(rev(empty)) = empty
   - rev(empty) = empty                                            (by SEQ.3)
   - empty = empty                                                 (by SEQ.3)
   CASE 1. finished

2. CASE: X1 = append(X2,X3)

   - rev(rev(append(X2,X3))) = append(X2,X3)
   - rev(conc(rev(X3),append(X2,empty))) = append(X2,X3)           (by SEQ.4)
   - rev(conc(rev(X3),append(X2,empty))) = append(X2,rev(rev(X3)))
                                                              (by IND.HYP.[1])
   - rev(conc(X4,append(X2,empty))) = append(X2,rev(X4))      (by GENERALIZATION)

     COMPLETE CASE ANALYSIS on X4                                      => [2]
     (Generated from the rules for function conc)
```

```
2.  1. CASE: X4 = empty

       - rev(conc(empty,append(X2,empty))) = append(X2,rev(empty))
       - rev(append(X2,empty)) = append(X2,rev(empty))             (by SEQ.1)
       - conc(rev(empty),append(X2,empty)) = append(X2,rev(empty))  (by SEQ.4)
       - conc(empty,append(X2,empty)) = append(X2,rev(empty))       (by SEQ.3)
       - append(X2,empty) = append(X2,rev(empty))                   (by SEQ.1)
       - X2 = X2
         & empty = rev(empty)                                 (append removed)
       - empty = empty                                              (by SEQ.3)
       CASE 2.1. finished

2.  2. CASE: X4 = append(X9,X10)

       - rev(conc(append(X9,X10),append(X2,empty))) =
           append(X2,rev(append(X9,X10)))
       - rev(append(X9,conc(X10,append(X2,empty)))) =
           append(X2,rev(append(X9,X10)))                           (by SEQ.2)
       - conc(rev(conc(X10,append(X2,empty))),append(X9,empty)) =
           append(X2,rev(append(X9,X10)))                           (by SEQ.4)
       - conc(rev(conc(X10,append(X2,empty))),append(X9,empty)) =
           append(X2,conc(rev(X10),append(X9,empty)))               (by SEQ.4)
       - conc(append(X2,rev(X10)),append(X9,empty)) =
           append(X2,conc(rev(X10),append(X9,empty)))         (by IND.HYP.[2])
       - append(X2,conc(rev(X10),append(X9,empty))) =
           append(X2,conc(rev(X10),append(X9,empty)))               (by SEQ.2)
       CASE 2.2. finished

THEOREM PROVED

STATISTICS                           THEOREM
  Lemma applications        :          0
  Generalizations           :          1
  Induction hypotheses used :          2
  Number of case analyses   :          2
  Total number of cases     :          4
  Decomposition steps       :          1
  Rewrite rule applications :         12
  CPU-time used (in msec)   :        584
```

Appendix E.2: Commutativity of Addition on ℕ

One of the best-known standard examples for inductive theorem provers: Specifying the addition of natural numbers only with two axioms and proving that this is already enough to deduce the validity of the commutativity.

To avoid the built-in definition of the addition in the standard type `NAT` (which is specified with 4 rules and of which the AC property is already known), we use our own simple specification of the natural numbers called `NAT_TRIVIAL`. Here we have defined “our” version of the addition `add2`.

First we prove the theorem `COMM` without AC information for `add2` to see the long proof. Then we tell TIP that `add2` has the AC property and repeat the proof. In the second case the proof is short but not very informative. For this purpose and to show the dealing with TIP in the automatic “mode” we list the whole log of this proof session:

```
TIP System Version 3.0 SUN 4.1/UNIX

Reading standard type declarations.

Reading input file nat_triv.

STEP 1: Syntax check
Reading type NAT_TRIVIAL.
Reading theorem COMM.
Reading theorem ASS.
No syntax errors detected.

STEP 2: Property check
noetherian NAT_TRIVIAL -- from input file
complete NAT_TRIVIAL -- from input file

STEP 3: Interactive induction prover
Preprocessing the axioms.

TIP> dir
Types:
BOOL                      NAT                       NAT_TRIVIAL


Theorems:
COMM                      ASS
```

TIP> *list NAT_TRIVIAL*

```
 type NAT_TRIVIAL

   sort Nat
   cons zero: Nat,
        succ: (Nat)Nat
   func add2: (Nat,Nat)Nat

 axioms all (x, y: Nat)

   {1} add2(zero,x) -> x,
   {2} add2(succ(x),y) -> succ(add2(x,y))

 endoftype
```

TIP> *list comm*

```
 theorem COMM

   basedon NAT_TRIVIAL

   forall x, y: Nat

 goals

   add2(x,y) = add2(y,x)

 endoftheorem
```

TIP> *prove COMM*

```
# CURRENT THEOREM: NO. 1
add2(X1,X2) = add2(X2,X1)
```

TIP/PROOF> *go*

 # THEOREM NO. 1 PROVED BY SYSTEM.

To make the proof listing a little bit shorter we use the **/COMPRESS** option. In this case several proof steps are grouped in one line. Of course, case analyses can not be grouped with other proof steps.

TIP/PROOF> *show proof /compress*

```
TIP System Version 3.0 SUN 4.1/UNIX                               Listing_of_Proof


PROOF OF THEOREM

 add2(X1,X2) = add2(X2,X1)

 COMPLETE CASE ANALYSIS on X1                                              => [1]
 (Generated from the rules for function add2)

1. CASE: X1 = zero

   - add2(zero,X2) = add2(X2,zero)
   - X2 = add2(X2,zero)                                        (by NAT_TRIVIAL.1)

     COMPLETE CASE ANALYSIS on X2                                          => [2]
     (Generated from the rules for function add2)

1.  1. CASE: X2 = zero

       - zero = add2(zero,zero)
       - zero = zero                                           (by NAT_TRIVIAL.1)
       CASE 1.1. finished

1.  2. CASE: X2 = succ(X10)

       - succ(X10) = add2(succ(X10),zero)
       - X10 = X10                (by NAT_TRIVIAL.2, succ removed, IND.HYP.[2])
       CASE 1.2. finished

2. CASE: X1 = succ(X4)

   - add2(succ(X4),X2) = add2(X2,succ(X4))
   - succ(add2(X2,X4)) = add2(X2,succ(X4))       (by NAT_TRIVIAL.2, IND.HYP.[1])

     COMPLETE CASE ANALYSIS on X2                                          => [3]
     (Generated from the rules for function add2)
```

```
2.  1. CASE: X2 = zero

       - succ(add2(zero,X4)) = add2(zero,succ(X4))
       - succ(X4) = succ(X4)                   (by NAT_TRIVIAL.1, NAT_TRIVIAL.1)
       CASE 2.1. finished

2.  2. CASE: X2 = succ(X28)

       - succ(add2(succ(X28),X4)) = add2(succ(X28),succ(X4))
       - add2(X28,succ(X4)) = add2(X28,succ(X4))              (by NAT_TRIVIAL.2,
                                        NAT_TRIVIAL.2, succ removed, IND.HYP.[3])
       CASE 2.2. finished

THEOREM PROVED

STATISTICS                          THEOREM
  Lemma applications        :           0
  Generalizations           :           0
  Induction hypotheses used :           3
  Number of case analyses   :           3
  Total number of cases     :           6
  Decomposition steps       :           2
  Rewrite rule applications :           8
  CPU-time used (in msec)   :        1350

TIP/PROOF> exit
```

The following command informs the TIP system that the function `add2` should have the AC property from now on. Then we repeat the proof of the theorem `COMM`:

```
TIP> set function ac add2
TIP> prove COMM
# CURRENT THEOREM: NO. 1
add2(X1,X2) = add2(X2,X1)

TIP/PROOF> go
# THEOREM NO. 1 PROVED BY SYSTEM.

TIP/PROOF>  show proof /nostatistic
TIP System Version 3.0 SUN 4.1/UNIX                                 Listing_of_Proof


PROOF OF THEOREM

 add2(X1,X2) = add2(X2,X1)
 add2(X1,X2) = add2(X2,X1)                                              (by AC-equal)


THEOREM PROVED

TIP/PROOF> exit
TIP> exit
```

Appendix E.3: Even or Odd

In this example we want to prove a conditional theorem. First we specify the properties even and odd for the natural numbers and then we give a theorem which states “every number which is not even has to be odd”.

```
 type EVEN

   basedon NAT,BOOL
   func even: (Nat)Bool,  { true if number is even }
        odd: (Nat)Bool    { true if number is odd }

 axioms all (n: Nat)
   {1} even(zero) -> true,
   {2} even(succ(zero)) -> false,
   {3} even(succ(succ(n))) -> even(n),
   {4} odd(zero) -> false,
   {5} odd(succ(zero)) -> true,
   {6} odd(succ(succ(n))) -> odd(n),
 endoftype


 theorem EVEN_ODD
   basedon EVEN,BOOL,NAT

   forall x, y: Nat, b: Bool
 goals
   even(x) = false => odd(x) = true
 endoftheorem
```

In the following proof listing you should notice the solution of a goal by a contradiction in its conditions and the application of a conditional induction hypothesis:

```
TIP System Version 3.0 SUN 4.1/UNIX                              Listing_of_Proof

PROOF OF THEOREM

 even(X1) = false => odd(X1) = true

 COMPLETE CASE ANALYSIS on X1                                              => [1]
 (Generated from the rules for function odd)

1. CASE: X1 = zero

   - even(zero) = false => odd(zero) = true
   - even(0) = false => false = true                                 (by EVEN.4)
   - true = false => false = true                                    (by EVEN.1)
   CASE 1. finished by contradiction in premises.

2. CASE: X1 = succ(zero)

   - even(succ(zero)) = false => odd(succ(zero)) = true
   - even(1) = false => true = true                                  (by EVEN.5)
   CASE 2. finished by trivial conclusion.

3. CASE: X1 = succ(succ(X2))

   - even(succ(succ(X2))) = false => odd(succ(succ(X2))) = true
   - even(succ(succ(X2))) = false => odd(X2) = true                  (by EVEN.6)
   - even(X2) = false => odd(X2) = true                              (by EVEN.3)
   - even(X2) = false => true = true
     & even(X2) = false => even(X2) = false                       (by IND.HYP.[1])
   CASE 3. finished by trivial implication and trivial conclusion.

THEOREM PROVED

STATISTICS                        THEOREM
  Lemma applications        :          0
  Generalizations           :          0
  Induction hypotheses used :          1
  Number of case analyses   :          1
  Total number of cases     :          3
  Decomposition steps       :          0
  Rewrite rule applications :          5
  CPU-time used (in msec)   :        101

TIP/PROOF>
```

Appendix E.4: Next Higher Even Number

The following example shows a proof where explicit condition handling is necessary. We want a function `f` which returns the next even number which is greater or equal than the argument. It is defined by two conditional axioms which only check if the argument is already an even number or increase it by one otherwise.

We show a listing of the whole proof session because this proof needs a case analysis in the conditions. Therefore this proof can be done only in an interactive way (up to now).

```
TIP System Version 3.0 SUN 4.1/UNIX

Reading standard type declarations.

Reading input file higher_even.

STEP 1: Syntax check
Reading type F_EVEN.
Reading theorem F_FF.
No syntax errors detected.

STEP 2: Property check
noetherian F_EVEN -- from input file
complete F_EVEN -- from input file

STEP 3: Interactive induction prover
Preprocessing the axioms.
```

TIP> dir

<u>Types:</u>
```
BOOL                        NAT                        F_EVEN
```

<u>Theorems:</u>
```
F_FF
```

TIP> *list F_EVEN*

```
 type F_EVEN

   basedon NAT,BOOL

   func even: (Nat)Bool,
        f: (Nat)Nat

 axioms all (x: Nat)

   {1} even(0) -> true,
   {2} even(1) -> false,
   {3} even(succ(succ(x))) -> even(x),
   {4} even(x) = true => f(x) -> x,
   {5} even(x) = false => f(x) -> succ(x)

 endoftype
```

TIP> *list F_FF*

```
 theorem F_FF

   basedon F_EVEN,NAT

   forall n: Nat

 goals

   f(n) = f(f(n))

 endoftheorem
```

TIP> *set param interactive*

<u>SYSTEM PARAMETERS SET FOR INTERACTIVE PROVING:</u>
```
USAGE OF IND. HYPOTHESES = INTERACTIVE (decided by user)
USAGE OF LEMMATA = INTERACTIVE (decided by user)
USAGE OF REWRITE RULES = CONTROLLED (ask only if rule is non-terminating)
GENERALIZATION = INTERACTIVE ONLY FOR MULTIPLE TERMS
STRATEGY FOR INDUCTION PROOF = DF (depth-first search)
REDEXSELECTION FOR CASE ANALYSIS = INTERACTIVE (decided by user)
CASE ANALYSIS = CONTEXT + VARS (by narrowing and for single variables)
CYCLE LIMIT = NO_CYCLES (no check for cycle-used applications)
```

TIP> *prove F_FF*
```
# CURRENT THEOREM: NO. 1
f(X1) = f(f(X1))
```

TIP/PROOF>
```
* CURRENT GOAL: NO. 1
f(X1) = f(f(X1))


* TRY SIMPLIFICATION OF GOAL NO. 1.
  GOAL IS SIMPLIFIED.
? POSSIBLE APPLICATION OF A CONDITIONAL REWRITE RULE: Rule F_EVEN.4:
   even(X1) = true => f(X1) -> X1
? AT
```
f(X1) = f(f(X1))

? Apply this conditional rewrite rule? (y/n/q) *n*
```
? POSSIBLE APPLICATION OF A CONDITIONAL REWRITE RULE: Rule F_EVEN.5:
   even(X1) = false => f(X1) -> succ(X1)
? AT
```
f(X1) = f(f(X1))

? Apply this conditional rewrite rule? (y/n/q) *n*

? POSSIBLE GENERALIZATION:
f(X1) = f(**f(X1)**)

? Generalizing? (y/n/q) *n*

```
? POSSIBLE INDUCTION REDEX:
f(X1) = f(f(X1))


? Try induction for this redex? (y/n/q) n
```

The two possible applications of conditional rewrite rules above indicate that enriching the conditions and doing a case analysis for the term `even(X1)` should be a useful step:

```
TIP/PROOF> enter condition 1
GOAL NO. 1:
f(X1) = f(f(X1))
 WITH (BINDINGS)
[]

> PLEASE ENTER THE TERM WHICH SHOULD BE USED FOR A NEW CASE ANALYSIS:
Insert (partial) equation. Use '?' for help or '.ABORT' to abort theorem
TERM: ?left side?
>> even X1
THIS TERM WILL BE USED FOR A NEW CASE ANALYSIS IN THE CONDITIONS:
even(X1)
? Do you want to do this? (y/n) y

* ADDING THE EQUATION  even(X1) = X3  TO THE CONDITIONS OF GOAL 1
* DOING A COMPLETE CASE ANALYSIS FOR GOAL NO. 1
* over variable X3
*   1. CASE: X3\true  [NEW GOAL: NO. 2]
*   2. CASE: X3\false  [NEW GOAL: NO. 3]

* TRY SIMPLIFICATION OF GOAL NO. 2.
*   GOAL REWRITTEN USING RULE F_EVEN.4
*   GOAL REWRITTEN USING RULE F_EVEN.4
*   GOAL REWRITTEN USING RULE F_EVEN.4
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 2 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 3.
*   GOAL REWRITTEN USING RULE F_EVEN.5
*   GOAL REWRITTEN USING RULE F_EVEN.5
  GOAL IS SIMPLIFIED.
TIP/PROOF>
* CURRENT GOAL: NO. 3
even(X1) = false => succ(X1) = f(succ(X1))

? POSSIBLE APPLICATION OF A CONDITIONAL REWRITE RULE: Rule F_EVEN.4:
   even(X1) = true => f(X1) -> X1
? AT
even(X1) = false => succ(X1) = f(succ(X1))

? Apply this conditional rewrite rule? (y/n/q) y
* USING THE CONDITIONAL REWRITERULE F_EVEN.4
   even(X1) = true => f(X1) -> X1
*** 1. premesis transformed to goal. [NEW GOAL: No. 4]
even(X1) = false => even(succ(X1)) = true

*** conclusion transformed to goal. [NEW GOAL: No. 5]
even(X1) = false => succ(X1) = succ(X1)


* TRY SIMPLIFICATION OF GOAL NO. 4.

* TRY SIMPLIFICATION OF GOAL NO. 5.
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 5 SOLVED.
```

A look at the proof tree can show us which branches of our proof are still unsolved:

```
TIP/PROOF> show tree
Proof Tree:

 GOAL 1  ACTIVE
   using COMPLETE CASE ANALYSIS
    1. Case:  X3\true
     GOAL 2  SUCCESS
    2. Case:  X3\false
     GOAL 3  ACTIVE  <- CURRENT
       using CONDITIONAL REWRITE RULE F_EVEN.4
        1. Premise:
         GOAL 4  ACTIVE
        Conclusion:
         GOAL 5  SUCCESS

TIP/PROOF>
* CURRENT GOAL: NO. 4
```

```
even(X1) = false => even(succ(X1)) = true


? POSSIBLE INDUCTION REDEX:
even(X1) = false => even(succ(X1)) = true



? Try induction for this redex? (y/n/q) y
* DOING A COMPLETE CASE ANALYSIS FOR GOAL NO. 4
* over variable X1 (generated by the rules of function even)
*   1. CASE: X1\0  [NEW GOAL: NO. 6]
*   2. CASE: X1\1  [NEW GOAL: NO. 7]
*   3. CASE: X1\succ(succ(X5))  [NEW GOAL: NO. 8]

* TRY SIMPLIFICATION OF GOAL NO. 6.
*   GOAL REWRITTEN USING RULE F_EVEN.2
*   GOAL REWRITTEN USING RULE F_EVEN.1
* CONTRADICTION IN PREMISES DETECTED => CLAUSE IS TRUE.
* THE IRREDUCIBLE SYMBOLS ARE >>> true <<< AND >>> false <<<
*   Goal NO. 6 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 7.
*   GOAL REWRITTEN USING RULE F_EVEN.3
*   GOAL REWRITTEN USING RULE F_EVEN.1
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 7 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 8.
*   GOAL REWRITTEN USING RULE F_EVEN.3
*   GOAL REWRITTEN USING RULE F_EVEN.3
  GOAL IS SIMPLIFIED.
TIP/PROOF>
* CURRENT GOAL: NO. 8
even(X5) = false => even(succ(X5)) = true

? POSSIBLE RULE USAGE: Hypothesis 2:
   even(X1) = false => even(succ(X1)) = true
? AT
even(X5) = false => even(succ(X5)) = true


? Apply this rule? (y/n/q) y
* REWRITE STEP USING INDUCTION HYPOTHESIS 2
*** 1. premesis transformed to goal. [NEW GOAL: No. 9]
even(X5) = false => even(X5) = false

*** conclusion transformed to goal. [NEW GOAL: No. 10]
even(X5) = false => true = true


* TRY SIMPLIFICATION OF GOAL NO. 9.
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 9 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 10.
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 10 SOLVED.
# THEOREM NO. 1 PROVED BY SYSTEM.

TIP/PROOF> show proof
TIP System Version 3.0 SUN 4.1/UNIX                            Listing_of_Proof


PROOF OF THEOREM

 f(X1) = f(f(X1))
 even(X1) = X3 => f(X1) = f(f(X1))

 COMPLETE CASE ANALYSIS on X3                                               => [1]

1. CASE: X3 = true

   - even(X1) = true => X1 = f(f(X1))
   - even(X1) = true => X1 = f(X1)                                (by F_EVEN.4)
   - even(X1) = true => X1 = X1                                   (by F_EVEN.4)
   CASE 1. finished by trivial conclusion.

2. CASE: X3 = false

   - even(X1) = false => f(X1) = f(f(X1))
   - even(X1) = false => succ(X1) = f(f(X1))                      (by F_EVEN.5)
   - even(X1) = false => succ(X1) = f(succ(X1))                   (by F_EVEN.5)
   - even(X1) = false => succ(X1) = succ(X1)
     & even(X1) = false => even(succ(X1)) = true                   (by F_EVEN.4)
```

```
     COMPLETE CASE ANALYSIS on X1                                         => [2]
     (Generated from the rules for function even)

2.  1. CASE: X1 = 0

        - even(0) = false => even(succ(0)) = true
        - even(0) = false => false = true                          (by F_EVEN.2)
        - true = false => false = true                             (by F_EVEN.1)
        CASE 2.1. finished by contradiction in premises.

2.  2. CASE: X1 = 1

        - even(1) = false => even(succ(1)) = true
        - even(1) = false => even(0) = true                        (by F_EVEN.3)
        - even(1) = false => true = true                           (by F_EVEN.1)
        CASE 2.2. finished by trivial conclusion.

2.  3. CASE: X1 = succ(succ(X5))

        - even(succ(succ(X5))) = false => even(succ(succ(succ(X5)))) = true
        - even(succ(succ(X5))) = false => even(succ(X5)) = true     (by F_EVEN.3)
        - even(X5) = false => even(succ(X5)) = true                 (by F_EVEN.3)
        - even(X5) = false => true = true
          & even(X5) = false => even(X5) = false                   (by IND.HYP.[2])
        CASE 2.3. finished by trivial implication and trivial conclusion.

THEOREM PROVED

STATISTICS                          THEOREM
  Lemma applications        :          0
  Generalizations           :          0
  Induction hypotheses used :          1
  Number of case analyses   :          2
  Total number of cases     :          5
  Decomposition steps       :          0
  Rewrite rule applications :         11
  CPU-time used (in msec)   :        334

TIP/PROOF> exit
TIP> exit
```

Appendix E.5: Binomial Coefficient

In this example we want to prove a rule for the binomial coefficient. First we specify a function `binom` which calculates the binomial coefficient $\binom{n}{r}$ (also called "*n* choose *r*"). This coefficient is an abbreviation for $\binom{n}{r} = \frac{n!}{(n-r)!\,r!}$. We specify this definition with the three axioms below. The theorem `BINOM1` states that $\binom{n}{r} = 0$ if $r > n$. The proof of this theorem shows that automatic rewriting is not always useful. To apply the induction hypothesis we must stuff one occurrence of `succ` into the second argument of `add`. But this step is against the rewrite direction, thus, automatic rewriting would redo it immediately.

The necessary lemma `SUCC_LEMMA` is also specified in our input file and its is assumed for the following proof. Of course TIP can prove this theorem by just one rewrite step.

```
type BINOMIAL
  basedon BOOL,NAT

  func binom: (Nat,Nat)Nat  { binomial coefficient }

axioms all (x, y, n, i: Nat)
  {1} binom(zero,succ(x)) -> zero,
  {2} binom(x,zero) -> succ(zero),
  {3} binom(succ(x),succ(y)) -> add(binom(x,succ(y)),binom(x,y)),
endoftype


theorem BINOM1
  basedon NAT,BINOMIAL

  forall x, y: Nat

goals
  binom(x,add(x,succ(y))) = zero
endoftheorem
```

```
 theorem SUCC_LEMMA
   basedon NAT

   forall x, y: Nat

 goals
   succ(add(x,y)) = add(x,succ(y))
 endoftheorem
```

```
TIP System Version 3.0 SUN 4.1/UNIX                           Listing_of_Proof


CORRECTNESS OF LEMMA No. 1 ASSUMED BY USER.
   LEMMA No. 1: succ(add(X3,X4)) = add(X3,succ(X4))


PROOF OF THEOREM

 binom(X1,add(X1,succ(X2))) = zero

 COMPLETE CASE ANALYSIS on X1                                            => [1]
 (Generated from the rules for function add)

1. CASE: X1 = zero

   - binom(zero,add(zero,succ(X2))) = 0
   - binom(0,succ(add(0,X2))) = 0                                    (by NAT.4)
   - binom(0,succ(X2)) = 0                                           (by NAT.5)
   - 0 = 0                                                      (by BINOMEAL.1)
   CASE 1. finished

2. CASE: X1 = succ(X6)

   - binom(succ(X6),add(succ(X6),succ(X2))) = 0
   - binom(succ(X6),succ(add(X6,succ(X2)))) = 0                      (by NAT.6)
   - add(binom(X6,succ(add(X6,succ(X2)))),binom(X6,add(X6,succ(X2)))) = 0
                                                                (by BINOMEAL.3)
   - add(binom(X6,succ(add(X6,succ(X2)))),0) = 0               (by IND.HYP.[1])
   - binom(X6,succ(add(X6,succ(X2)))) = 0                            (by NAT.3)
   - binom(X6,add(X6,succ(succ(X2)))) = 0                      (by LEMMA No. 1)
   - 0 = 0                                                     (by IND.HYP.[1])
   CASE 2. finished

THEOREM PROVED

STATISTICS                           THEOREM
  Lemma applications        :           1
  Generalizations           :           0
  Induction hypotheses used :           2
  Number of case analyses   :           1
  Total number of cases     :           2
  Decomposition steps       :           0
  Rewrite rule applications :           6
  CPU-time used (in msec)   :         333
```

Appendix E.6: The "if" Function

Here we want to deal with an unconditionally defined if-function. Therefore, if the boolean argument of the if-function is a term t, then we can either wait until t can be evaluated to true or false or we add the equation $t = x$ to the conditions and do a case analysis over the new variable x.

The theorem `IF_EQU` states that $x + x = \begin{cases} (x+x) & \text{if } x \neq y \\ (x+y) & \text{if } x = y \end{cases}$.

Again we present the full listing of the TIP-session to demonstrate the interactive way of dealing with the prover for conditional proofs:

```
TIP System Version 3.0 SUN 4.1/UNIX

Reading standard type declarations.

Reading input file if2.

STEP 1: Syntax check
Reading type IF.
Reading theorem IF_EQU.
No syntax errors detected.
```

STEP 2: Property check
noetherian IF -- from input file
complete IF -- from input file

STEP 3: Interactive induction prover
Preprocessing the axioms.

TIP> dir
Types:
BOOL NAT IF

Theorems:
IF_EQU

TIP> *list if*

```
 type IF

   basedon NAT,BOOL

   func if: (Bool,Nat,Nat)Nat

 axioms all (x, y: Nat)

   {1} if(true,x,y) -> x,
   {2} if(false,x,y) -> y

 endoftype
```

TIP> *list IF_EQU*

```
 theorem IF_EQU

   basedon IF,NAT,BOOL

   forall x, y: Nat

 goals

   if(equal_Nat(x,y),add(x,y),add(x,x)) = add(x,x)

 endoftheorem
```

TIP> *set param interactive*

```
SYSTEM PARAMETERS SET FOR INTERACTIVE PROVING:
USAGE OF IND. HYPOTHESES = INTERACTIVE (decided by user)
USAGE OF LEMMATA = INTERACTIVE (decided by user)
USAGE OF REWRITE RULES = CONTROLLED (ask only if rule is non-terminating)
GENERALIZATION = INTERACTIVE ONLY FOR MULTIPLE TERMS
STRATEGY FOR INDUCTION PROOF = DF (depth-first search)
REDEXSELECTION FOR CASE ANALYSIS = INTERACTIVE (decided by user)
CASE ANALYSIS = CONTEXT + VARS (by narrowing and for single variables)
CYCLE LIMIT = NO_CYCLES (no check for cycle-used applications)
```

TIP> prove IF_EQU
```
# CURRENT THEOREM: NO. 1
if(equal_Nat(X1,X2),add(X1,X2),add(X1,X1)) = add(X1,X1)
```

TIP/PROOF>
```
* CURRENT GOAL: NO. 1
if(equal_Nat(X1,X2),add(X1,X2),add(X1,X1)) = add(X1,X1)


* TRY SIMPLIFICATION OF GOAL NO. 1.
  GOAL IS SIMPLIFIED.

? POSSIBLE GENERALIZATION:
if(equal_Nat(X1,X2),add(X1,X2),add(X1,X1)) = add(X1,X1)


? Generalizing? (y/n/q) n
```

The generalization above is no good idea, because for `equal_Nat(X1,X2)=true` the goal would be `add(X1,X2) = X3`. But of course this is not true for universally quantified variables.

```
? POSSIBLE INDUCTION REDEX:
if(equal_Nat(X1,X2),add(X1,X2),add(X1,X1)) = add(X1,X1)


? Try induction for this redex? (y/n/q) q
```

A case analysis is no good idea here, because the `succ`'s of the second and third argument of `if` will be stuck inside the term. To remove them and to apply the induction hypothesis you would need additionally lemmas. Therefore we try to enrich the conditions of this goal:

```
TIP/PROOF> enter condition 1
GOAL NO. 1:
if(equal_Nat(X1,X2),add(X1,X2),add(X1,X1)) = add(X1,X1)
 WITH (BINDINGS)
[]

> PLEASE ENTER THE TERM WHICH SHOULD BE USED FOR A NEW CASE ANALYSIS:
Insert (partial) equation. Use `?` for help or `.ABORT` to abort theorem
TERM: ?left side?
>> equal_Nat(X1,X2)
THIS TERM WILL BE USED FOR A NEW CASE ANALYSIS IN THE CONDITIONS:
equal_Nat(X1,X2)
? Do you want to do this? (y/n) y

* ADDING THE EQUATION  equal_Nat(X1,X2) = X21  TO THE CONDITIONS OF GOAL 1
* DOING A COMPLETE CASE ANALYSIS FOR GOAL NO. 1
* over variable X21
*   1. CASE: X21\true  [NEW GOAL: NO. 2]
*   2. CASE: X21\false  [NEW GOAL: NO. 3]

* TRY SIMPLIFICATION OF GOAL NO. 2.
*   GOAL REWRITTEN USING RULE IF.1
  GOAL IS SIMPLIFIED.

* TRY SIMPLIFICATION OF GOAL NO. 3.
*   GOAL REWRITTEN USING RULE IF.2
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 3 SOLVED.
TIP/PROOF> show goal 2
GOAL NO. 2 [STATE IS ACTIVE] :
equal_Nat(X1,X2) = true => add(X1,X2) = add(X1,X1)
 WITH (BINDINGS)
[X21 = true]
TIP/PROOF>
* CURRENT GOAL: NO. 2
equal_Nat(X1,X2) = true => add(X1,X2) = add(X1,X1)

? POSSIBLE INDUCTION REDEX:
equal_Nat(X1,X2) = true => add(X1,X2) = add(X1,X1)

? Try induction for this redex? (y/n/q) n

? POSSIBLE INDUCTION REDEX:
equal_Nat(X1,X2) = true => add(X1,X2) = add(X1,X1)

? Try induction for this redex? (y/n/q) n

? POSSIBLE INDUCTION REDEX:
equal_Nat(X1,X2) = true => add(X1,X2) = add(X1,X1)


? Try induction for this redex? (y/n/q) y
```

We choose this case analysis, because the equality of `X1` and `X2` is the decisive criterion. If these two variables are greater than zero then we can apply the induction hypothesis (see case 4).

```
* DOING A COMPLETE CASE ANALYSIS FOR GOAL NO. 2
* over variables X1 and X2 (generated by the rules of function equal_Nat)
*   1. CASE: X1\X32, X2\X32  [NEW GOAL: NO. 4]
*   2. CASE: X1\zero, X2\succ(X33)  [NEW GOAL: NO. 5]
*   3. CASE: X1\succ(X34), X2\zero  [NEW GOAL: NO. 6]
*   4. CASE: X1\succ(X35), X2\succ(X36)  [NEW GOAL: NO. 7]

* TRY SIMPLIFICATION OF GOAL NO. 4.
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 4 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 5.
*   GOAL REWRITTEN USING RULE NAT.4
*   GOAL REWRITTEN USING RULE NAT.5
*   GOAL REWRITTEN USING RULE NAT.26
* CONTRADICTION IN PREMISES DETECTED => CLAUSE IS TRUE.
* THE IRREDUCIBLE SYMBOLS ARE >>> false <<< AND >>> true <<<
*   Goal NO. 5 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 6.
*   GOAL REWRITTEN USING RULE NAT.3
*   GOAL REWRITTEN USING RULE NAT.4
*   GOAL REWRITTEN USING RULE NAT.6
*   GOAL REWRITTEN USING RULE NAT.27
```

```
* CONTRADICTION IN PREMISES DETECTED => CLAUSE IS TRUE.
* THE IRREDUCIBLE SYMBOLS ARE >>> false <<< AND >>> true <<<
*   Goal NO. 6 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 7.
*   GOAL REWRITTEN USING RULE NAT.4
*   GOAL REWRITTEN USING RULE NAT.6
*   GOAL REWRITTEN USING RULE NAT.4
*   GOAL REWRITTEN USING RULE NAT.6
*   GOAL REWRITTEN USING RULE NAT.28
  GOAL IS SIMPLIFIED.
TIP/PROOF>
* CURRENT GOAL: NO. 7
equal_Nat(X35,X36) = true => add(X35,X36) = add(X35,X35)

? POSSIBLE RULE USAGE: Hypothesis 2:
   equal_Nat(X1,X2) = true => add(X1,X2) = add(X1,X1)
? AT
equal_Nat(X35,X36) = true => add(X35,X36) = add(X35,X35)


? Apply this rule? (y/n/q) y
* REWRITE STEP USING INDUCTION HYPOTHESIS 2
*** 1. premesis transformed to goal. [NEW GOAL: No. 8]
equal_Nat(X35,X36) = true => equal_Nat(X35,X36) = true

*** conclusion transformed to goal. [NEW GOAL: No. 9]
equal_Nat(X35,X36) = true => add(X35,X35) = add(X35,X35)


* TRY SIMPLIFICATION OF GOAL NO. 8.
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 8 SOLVED.

* TRY SIMPLIFICATION OF GOAL NO. 9.
*   TRIVIAL CONCLUSION; CLAUSE REMOVED.
*   Goal NO. 9 SOLVED.
# THEOREM NO. 1 PROVED BY SYSTEM.

TIP/PROOF> show proof
TIP System Version 3.0 SUN 4.1/UNIX                        Listing_of_Proof


PROOF OF THEOREM

 if(equal_Nat(X1,X2),add(X1,X2),add(X1,X1)) = add(X1,X1)
 equal_Nat(X1,X2) = X21 =>
    if(equal_Nat(X1,X2),add(X1,X2),add(X1,X1)) = add(X1,X1)

 COMPLETE CASE ANALYSIS on X21                                        => [1]

1. CASE: X21 = true

   - equal_Nat(X1,X2) = true => if(true,add(X1,X2),add(X1,X1)) = add(X1,X1)
   - equal_Nat(X1,X2) = true => add(X1,X2) = add(X1,X1)              (by IF.1)

     COMPLETE CASE ANALYSIS on X1 and X2                              => [2]
     (Generated from the rules for function equal_Nat)

1.   1. CASE: X1 = X32, X2 = X32

       - equal_Nat(X32,X32) = true => add(X32,X32) = add(X32,X32)
       CASE 1.1. finished

1.   2. CASE: X1 = zero, X2 = succ(X33)

       - equal_Nat(zero,succ(X33)) = true =>
          add(zero,succ(X33)) = add(zero,zero)
       - equal_Nat(0,succ(X33)) = true => succ(add(0,X33)) = 0        (by NAT.4)
       - equal_Nat(0,succ(X33)) = true => succ(X33) = 0               (by NAT.5)
       - false = true => succ(X33) = 0                               (by NAT.26)
* CONTRADICTION IN PREMISES DETECTED => CLAUSE IS TRUE.
* THE IRREDUCIBLE SYMBOLS ARE >>> false <<< AND >>> true <<<
       CASE 1.2. finished

1.   3. CASE: X1 = succ(X34), X2 = zero

       - equal_Nat(succ(X34),zero) = true =>
          add(succ(X34),zero) = add(succ(X34),succ(X34))
       - equal_Nat(succ(X34),0) = true => succ(X34) = add(succ(X34),succ(X34))
                                                                      (by NAT.3)
       - equal_Nat(succ(X34),0) = true => succ(X34) = succ(add(succ(X34),X34))
                                                                      (by NAT.4)
       - equal_Nat(succ(X34),0) = true => X34 = add(succ(X34),X34)
                                                                  (succ removed)
       - equal_Nat(succ(X34),0) = true => X34 = succ(add(X34,X34))    (by NAT.6)
       - false = true => X34 = succ(add(X34,X34))                    (by NAT.27)
```

```
* CONTRADICTION IN PREMISES DETECTED => CLAUSE IS TRUE.
* THE IRREDUCIBLE SYMBOLS ARE >>> false <<< AND >>> true <<<
       CASE 1.3. finished

1.  4. CASE: X1 = succ(X35), X2 = succ(X36)

       - equal_Nat(succ(X35),succ(X36)) = true =>
          add(succ(X35),succ(X36)) = add(succ(X35),succ(X35))
       - equal_Nat(succ(X35),succ(X36)) = true =>
          succ(add(succ(X35),X36)) = add(succ(X35),succ(X35))          (by NAT.4)
       - equal_Nat(succ(X35),succ(X36)) = true =>
          succ(succ(add(X35,X36))) = add(succ(X35),succ(X35))          (by NAT.6)
       - equal_Nat(succ(X35),succ(X36)) = true =>
          succ(succ(add(X35,X36))) = succ(add(succ(X35),X35))          (by NAT.4)
       - equal_Nat(succ(X35),succ(X36)) = true =>
          succ(add(X35,X36)) = add(succ(X35),X35)                  (succ removed)
       - equal_Nat(succ(X35),succ(X36)) = true =>
          succ(add(X35,X36)) = succ(add(X35,X35))                      (by NAT.6)
       - equal_Nat(succ(X35),succ(X36)) = true => add(X35,X36) = add(X35,X35)
                                                                   (succ removed)
       - equal_Nat(X35,X36) = true => add(X35,X36) = add(X35,X35)     (by NAT.28)
       - equal_Nat(X35,X36) = true => add(X35,X35) = add(X35,X35)
         & equal_Nat(X35,X36) = true => equal_Nat(X35,X36) = true
                                                                 (by IND.HYP.[2])
       CASE 1.4. finished

2. CASE: X21 = false

   - equal_Nat(X1,X2) = false => if(false,add(X1,X2),add(X1,X1)) = add(X1,X1)
   - equal_Nat(X1,X2) = false => add(X1,X1) = add(X1,X1)                (by IF.2)
   CASE 2. finished

THEOREM PROVED

STATISTICS                          THEOREM
  Lemma applications        :            0
  Generalizations           :            0
  Induction hypotheses used :            1
  Number of case analyses   :            2
  Total number of cases     :            6
  Decomposition steps       :            3
  Rewrite rule applications :           14
  CPU-time used (in msec)   :          399

TIP/PROOF> exit
TIP> exit
```

Appendix E.7: Binary Sorting

First we specify a sequence of the binary symbols `O` and `L`, then we define a function `bsort` for sorting such a sequence and finally we have a “predicate” `issorted` which checks if a given sequence is in a sorted state.

```
type BINSEQ

  sort Bin,BSeq
  cons O: Bin,
       L: Bin,
       empty: BSeq,
       app: (Bin,BSeq)BSeq  { append one element }

endoftype

type SORT
  basedon BINSEQ,NAT,BOOL

  func bsort: (BSeq)BSeq,    { binary sorting }
       e: (BSeq)BSeq,        { aux. function for inserting L's }
       issorted: (BSeq)Bool  { check is seq. is sorted }

axioms all (s: BSeq, b: Bin)
  {1}  bsort(empty) -> empty,
  {2}  bsort(app(O,s)) -> app(O,bsort(s)),
  {3}  bsort(app(L,s)) -> e(bsort(s)),  {insert L at the right place}
  {4}  e(empty) -> app(L,empty),
  {5}  e(app(O,s)) -> app(O,e(s)),
  {6}  e(app(L,s)) -> app(L,app(L,s)),
  {7}  issorted(empty) -> true,
  {8}  issorted(app(b,empty)) -> true,
  {9}  issorted(app(O,app(O,s))) -> issorted(app(O,s)),
  {10} issorted(app(L,app(L,s))) -> issorted(app(L,s)),
  {11} issorted(app(O,app(L,s))) -> issorted(app(L,s)),
  {12} issorted(app(L,app(O,s))) -> false
endoftype
```

Theorem SORTED

The theorem `SORTED` is a test whether the sorting algorithm works correctly. This is done by controlling the output of the function `bsort`.

```
theorem SORTED
  basedon BINSEQ,SORT,BOOL

  forall s: BSeq
goals

  issorted(bsort(s)) = true

endoftheorem
```

In the proof of `SORTED` needs two lemmas. These lemmas have to be entered by the user, but then they can by proved automatically by the TIP system. They are necessary for the cases 2 and 3 in the proof of the theorem to enable the application of the induction hypothesis. `Lemma No. 1` states that an `O` at first position of a sequence can be ignored when testing if a sequence is sorted, while `Lemma No. 2` states that the auxiliary function `e` can ignored at all when testing if a sequence is sorted.

```
TIP System Version 3.1 SUN 4.1/UNIX                              Listing of Proof

PROOF OF LEMMA No. 1

 issorted(app(O,X12)) = issorted(X12)


 COMPLETE CASE ANALYSIS on X12                                               => [1]

 1. CASE: X12 = empty

   - issorted(app(O,empty)) = issorted(empty)
   - true = issorted(empty)                                              (by SORT.8)
   - true = true                                                         (by SORT.7)
   CASE 1. finished.

 2. CASE: X12 = app(X13,X14)

   - issorted(app(O,app(X13,X14))) = issorted(app(X13,X14))


    COMPLETE CASE ANALYSIS on X13                                            => [2]
   (Generated from the rules for function issorted)

   2. 1. CASE: X13 = O

      - issorted(app(O,app(O,X14))) = issorted(app(O,X14))
      - issorted(app(O,X14)) = issorted(app(O,X14))                      (by SORT.9)
      CASE 2.1. finished.

   2. 2. CASE: X13 = L

      - issorted(app(O,app(L,X14))) = issorted(app(L,X14))
      - issorted(app(L,X14)) = issorted(app(L,X14))                     (by SORT.11)
      CASE 2.2. finished.

LEMMA No. 1 PROVED


PROOF OF LEMMA No. 2

 issorted(e(X2)) = issorted(X2)


 COMPLETE CASE ANALYSIS on X2                                                => [3]
 (Generated from the rules for function issorted)

 1. CASE: X2 = empty

   - issorted(e(empty)) = issorted(empty)
   - issorted(app(L,empty)) = issorted(empty)                            (by SORT.4)
   - true = issorted(empty)                                              (by SORT.8)
   - true = true                                                         (by SORT.7)
   CASE 1. finished.

 2. CASE: X2 = app(X5,empty)

   - issorted(e(app(X5,empty))) = issorted(app(X5,empty))
   - issorted(e(app(X5,empty))) = true                                   (by SORT.8)
```

```
 COMPLETE CASE ANALYSIS on X5                                     => [4]
(Generated from the rules for function e)

2. 1. CASE: X5 = O

   - issorted(e(app(O,empty))) = true
   - issorted(app(O,e(empty))) = true                         (by SORT.5)
   - issorted(app(O,app(L,empty))) = true                     (by SORT.4)
   - issorted(app(L,empty)) = true                           (by SORT.11)
   - true = true                                              (by SORT.8)
   CASE 2.1. finished.

2. 2. CASE: X5 = L

   - issorted(e(app(L,empty))) = true
   - issorted(app(L,app(L,empty))) = true                     (by SORT.6)
   - issorted(app(L,empty)) = true                           (by SORT.10)
   - true = true                                              (by SORT.8)
   CASE 2.2. finished.
```

```
3. CASE: X2 = app(O,app(O,X6))

  - issorted(e(app(O,app(O,X6)))) = issorted(app(O,app(O,X6)))
  - issorted(app(O,e(app(O,X6)))) = issorted(app(O,app(O,X6)))      (by SORT.5)
  - issorted(app(O,app(O,e(X6)))) = issorted(app(O,app(O,X6)))      (by SORT.5)
  - issorted(app(O,e(X6))) = issorted(app(O,app(O,X6)))             (by SORT.9)
  - issorted(app(O,e(X6))) = issorted(app(O,X6))                    (by SORT.9)
  - issorted(app(O,e(X6))) = issorted(e(app(O,X6)))            (by IND.HYP.[3])
  - issorted(app(O,e(X6))) = issorted(app(O,e(X6)))                 (by SORT.5)
  CASE 3. finished.

4. CASE: X2 = app(L,app(L,X7))

  - issorted(e(app(L,app(L,X7)))) = issorted(app(L,app(L,X7)))
  - issorted(app(L,app(L,app(L,X7)))) = issorted(app(L,app(L,X7)))  (by SORT.6)
  - issorted(app(L,app(L,X7))) = issorted(app(L,app(L,X7)))        (by SORT.10)
  CASE 4. finished.

5. CASE: X2 = app(O,app(L,X8))

  - issorted(e(app(O,app(L,X8)))) = issorted(app(O,app(L,X8)))
  - issorted(app(O,e(app(L,X8)))) = issorted(app(O,app(L,X8)))      (by SORT.5)
  - issorted(app(O,app(L,app(L,X8)))) = issorted(app(O,app(L,X8)))  (by SORT.6)
  - issorted(app(L,app(L,X8))) = issorted(app(O,app(L,X8)))        (by SORT.11)
  - issorted(app(L,X8)) = issorted(app(O,app(L,X8)))               (by SORT.10)
  - issorted(app(L,X8)) = issorted(app(L,X8))                      (by SORT.11)
  CASE 5. finished.

6. CASE: X2 = app(L,app(O,X9))

  - issorted(e(app(L,app(O,X9)))) = issorted(app(L,app(O,X9)))
  - issorted(app(L,app(L,app(O,X9)))) = issorted(app(L,app(O,X9)))  (by SORT.6)
  - issorted(app(L,app(O,X9))) = issorted(app(L,app(O,X9)))        (by SORT.10)
  CASE 6. finished.

LEMMA No. 2 PROVED
```

PROOF OF THEOREM

```
 issorted(bsort(X1)) = true


 COMPLETE CASE ANALYSIS on X1                                     => [5]
 (Generated from the rules for function bsort)

 1. CASE: X1 = empty

   - issorted(bsort(empty)) = true
   - issorted(empty) = true                                   (by SORT.1)
   - true = true                                              (by SORT.7)
   CASE 1. finished.

 2. CASE: X1 = app(O,X21)

   - issorted(bsort(app(O,X21))) = true
   - issorted(app(O,bsort(X21))) = true                       (by SORT.2)
   - issorted(bsort(X21)) = true                         (by LEMMA No. 1)
   - true = true                                         (by IND.HYP.[5])
   CASE 2. finished.

 3. CASE: X1 = app(L,X22)

   - issorted(bsort(app(L,X22))) = true
   - issorted(e(bsort(X22))) = true                           (by SORT.3)
   - issorted(bsort(X22)) = true                         (by LEMMA No. 2)
   - true = true                                         (by IND.HYP.[5])
   CASE 3. finished.
```

```
THEOREM PROVED

STATISTICS                     LEMMA No. 1  LEMMA No. 2      THEOREM
  Lemma applications        :           0            0            2
  Generalizations           :           0            0            0
  Induction hypotheses used :           0            1            2
  Number of case analyses   :           2            2            1
  Total number of cases     :           4            8            3
  Decomposition steps       :           0            0            0
  Rewrite rule applications :           4           25            4
  CPU-time used (in msec)   :         266          217         1750
```

Theorem SORTSORT

The second theorem for binary sorting, named SORTSORT, checks whether sorting a sorted sequence leaves this sequence unchanged.

```
theorem SORTSORT
  basedon BINSEQ,SORT

  forall s: BSeq
goals

  bsort(bsort(s)) = bsort(s)

endoftheorem
```

In the proof of SORTSORT a very tricky lemma is found automatically by the TIP system. After the inverse application of the hypothesis in case three a generalization is possible. By these two steps the important equation `bsort (e(X6)) = e(bsort(X6))` is generated and proved by induction afterwards. In a former version of the prove this equation had to be given to TIP as an auxiliary lemma.

In version 3.0 of TIP the improved generalization and mainly the heuristics for pruning the proof tree (section 4.6.3) make it possible to do this prove on a normal workstation without memory problems.

```
TIP System Version 3.1 SUN 4.1/UNIX                           Listing of Proof

PROOF OF THEOREM

 bsort(bsort(X1)) = bsort(X1)


 COMPLETE CASE ANALYSIS on X1                                          => [1]
 (Generated from the rules for function bsort)

 1. CASE: X1 = empty

   - bsort(bsort(empty)) = bsort(empty)
   - bsort(empty) = bsort(empty)                                  (by SORT.1)
   CASE 1. finished.

 2. CASE: X1 = app(O,X2)

   - bsort(bsort(app(O,X2))) = bsort(app(O,X2))
   - bsort(app(O,bsort(X2))) = bsort(app(O,X2))                   (by SORT.2)
   - app(O,bsort(bsort(X2))) = bsort(app(O,X2))                   (by SORT.2)
   - app(O,bsort(bsort(X2))) = app(O,bsort(X2))                   (by SORT.2)
   - bsort(X2) = bsort(X2)                                    (by IND.HYP.[1])
   CASE 2. finished.

 3. CASE: X1 = app(L,X3)

   - bsort(bsort(app(L,X3))) = bsort(app(L,X3))
   - bsort(e(bsort(X3))) = bsort(app(L,X3))                       (by SORT.3)
   - bsort(e(bsort(X3))) = e(bsort(X3))                           (by SORT.3)
   - bsort(e(bsort(X3))) = e(bsort(bsort(X3)))                (by IND.HYP.[1])
   - bsort(e(X6)) = e(bsort(X6))                           (by GENERALIZATION)

    COMPLETE CASE ANALYSIS on X6                                       => [2]
   (Generated from the rules for function e)

   3. 1. CASE: X6 = empty

      - bsort(e(empty)) = e(bsort(empty))
      - bsort(app(L,empty)) = e(bsort(empty))                     (by SORT.4)
      - e(bsort(empty)) = e(bsort(empty))                         (by SORT.3)
      CASE 3.1. finished.
```

3. 2. CASE: X6 = app(O,X7)

```
- bsort(e(app(O,X7))) = e(bsort(app(O,X7)))
- bsort(app(O,e(X7))) = e(bsort(app(O,X7)))            (by SORT.5)
- app(O,bsort(e(X7))) = e(bsort(app(O,X7)))            (by SORT.2)
- app(O,bsort(e(X7))) = e(app(O,bsort(X7)))            (by SORT.2)
- app(O,bsort(e(X7))) = app(O,e(bsort(X7)))            (by SORT.5)
- e(bsort(X7)) = e(bsort(X7))                      (by IND.HYP.[2])
CASE 3.2. finished.
```

3. 3. CASE: X6 = app(L,X8)

```
- bsort(e(app(L,X8))) = e(bsort(app(L,X8)))
- bsort(app(L,app(L,X8))) = e(bsort(app(L,X8)))        (by SORT.6)
- e(bsort(app(L,X8))) = e(bsort(app(L,X8)))            (by SORT.3)
CASE 3.3. finished.
```

THEOREM PROVED

STATISTICS		**THEOREM**
Lemma applications	:	0
Generalizations	:	1
Induction hypotheses used	:	3
Number of case analyses	:	2
Total number of cases	:	6
Decomposition steps	:	0
Rewrite rule applications	:	14
CPU-time used (in msec)	:	15434

Index of TIP Commands

Index

W

X

Y