Some Theorem Proving Aids

Paul E. Black[1]* and Phillip J. Windley[2]

[1] National Institute of Standards and Technology, Gaithersburg, Maryland 20899
[2] Computer Science Department, Brigham Young University, Provo, Utah 84602-6576

Abstract. Theorem proving can be a very useful formal method. However it currently takes a lot of time and study to learn how to use a theorem prover, and proving even apparently simple theorems can be tedious. Theorem proving, and its benefits in software and hardware development, should be accepted more readily and widely if new users can do larger proofs of more complete models earlier in their training and with less work.
We present some generally applicable tools which we found helpful in formally verifying a secure web server. The first is a program to check goals for common mistakes arising indirectly from type inference. We also give tactics, or proof advancing routines, to simplify goals and handle assumptions. Finally we give tactics which prove goals by selecting assumptions to establish the goal or find a contradiction. These are another step to making theorem proving easier, increasing productive, and reducing unnecessary complication.

1 Introduction

Although proving theorems with mechanized support can be useful in many industrial developments, using a theorem prover can take a great deal of expertise. Many people consider theorem proving to be unrealistic because of the time to learn how to do proofs and the tedium of proofs. However more powerful tactics would allow users to "work at a high enough level to make the proof process practical" [3].

Unfortunately powerful tactics tend to be slow. In fact, Gödel's theorem assures us there is a limit to the power of automatic tactics [5, 6]. However as computers have gotten faster, it makes sense to automate more and more theorem proving, even using heuristics which may be time-consuming or not guaranteed to work. `MESON_TAC` [4] is an example. HOL traditionally has concentrated on efficient "building block" tactics rather than strong tactics as, say, PVS [7] or Isabelle [8].

We describe general tactics and a program we developed while verifying a secure web server [1]. Section 2 presents a small program to help analyze why a rewrite may fail on a particular goal. Section 3 gives several tactics to simplify

* This work was sponsored in part by the National Science Foundation under NSF grant MIP–9412581

goals. The last set of tactics, in Sect. 4, attempt to prove goals by grouping assumptions for other conversions. (Early versions of these sections are in [1].) Finally we report our use of these tactics and our conclusions in Sect. 6.

2 A Program to Analyze Rewrite Failures

Our first aid to theorem proving is a small program to analyze rewrite failures. One may make mistakes in assigning types or may misenter names, especially when developing specifications or formalizations. These errors may be especially frustrating when a rewrite fails for no apparent reason. We wrote a routine in SML to help analyze these situations.

`WHY_NOT()` examines the current, or top, goal and reports possible subtleties which may be preventing it from being proved. It searches for and reports the following situations.

1. Variables or constants with the same name, but different types.
2. Variables or constants with the same type and similar names. That is, possible typographical errors.
3. A variable and a constant with the same name and type.
4. Variables whose names are valid constants for that common types.

Such subtle differences are hard to spot and have wasted a lot of users' time finding them. Since types in HOL are often inferred, identifiers with the same name, but different types, are rare. But because HOL typically does not print types, the user may have a hard time finding the problem when it does occur. Also with the type inference, it is not unheard–of to mistype a variable or constant and not notice the problem for some time. For example, one researcher typed a goal similar to the following.

$$(\texttt{empty\_q} \Rightarrow \texttt{done}) \wedge (\neg\texttt{done} \wedge \texttt{started} \Rightarrow \neg\texttt{emtpy\_q})$$

Boolean types were inferred for all variables. It took several frustrating hours trying to prove the goal before realizing that `empty_q` had been mistyped as `emtpy_q`.

Even more difficult to find is if a constant and a variable have the same name and type. There is printed indication that something is a constant instead of a variable. Finally it is possible to create a numeric variable named 42 or a string variable named `"k"`. Again there is no way to distinguish these from constants except using the predicates `is_const` or `is_var`.

The implementation is straight forward. First all atoms (variables or constants) are extracted from the top goal and assumption list. Numeric constants, short string constants, and constants whose names are common operators, such as $\forall$ or $+$, are ignored. The "type" check reports atoms with the same name, but different types. The "spelling" check reports atoms which identical types and similar names. Names are similar if the initial character is the same and the rest differs by a single character replacement, deletion, or insertion or a single transposition. We require the initial characters to match to avoid reporting pairs like `xSize` and `ySize`. Names shorter than three characters are ignored, too, to reduce false reports. The "kind" check reports atoms with the same name and type where one is a constant and the other is a variable. Finally list of atoms is checked for variables which are numerals, strings, or common operators.

The following is a contrived, small example. The intent is to set a goal to prove $\mathtt{x} + 2 + \mathtt{abc} = \mathtt{abc} + 1 + \mathtt{y} + 1$ given $\mathtt{x} = \mathtt{y} \wedge \mathtt{abc} > 5$. However `abc` is mistyped as `abe`. More seriously since $=$ has lower precedence than $\wedge$, the hypothesis is associated as $\mathtt{x} = (\mathtt{y} \wedge (\mathtt{abc} > 5))$, so `x` and `y` are type boolean instead of natural numbers. The errors are pretty obvious here, but these kinds of errors are much harder to catch when the predicates are big or there are lots of assumptions. The goal looks like it could be proved by rewriting and arithmetic analysis (`ASM_REWRITE_TAC []` and `CONV_TAC ARITH_CONV`), but the "errors" prevent it.

```
- set_goal([x = y ∧ abc > 5],
          x + 2 + abc = abe + 1 + y + 1);

val it =
    Initial goal:
          x + 2 + abc = abe + 1 + y + 1
          ____________________________
                x = y ∧ abc > 5

- WHY_NOT();

The name y appears as both :num and :bool
The name x appears as both :num and :bool
Possible typo: abe and abc have the same type and similar names
```

Types in HOL are prefaced with a colon (`:num` and `:bool`). The user prompt is a dash and space (`- `). We give more details about HOL in App. A.

3 Tactics to Simplify Goals

Often the theorem to be proved is so complex it is hard for the novice to know where to start. These complex goals may arise in discharging obligations of theorems, so it may not even be immediately clear what the theorem means. This section gives tactics we have found helpful to simplify goals and take small, but significant, steps toward proving them. The tactics may even prove the goal.

3.1 General Simplification

Proofs in axiomatic semantics tend to carry conjunctions of many conditions. Inference rules often involve one extended condition implying another where most of the conditions can be trivially satisfied. So there are often goals similar to a $\wedge$ b $\wedge$ c $\wedge$ D $\Rightarrow$ a $\wedge$ b $\wedge$ c $\wedge$ E.

Sometimes a goal can be proved just by stripping quantifiers and implications, then rewriting with assumptions. We combine these steps into one tactic which we call STRIP_THEN_REWRITE_TAC. It is simply

```
val STRIP_THEN_REWRITE_TAC =
    REPEAT STRIP_TAC THEN ASM_REWRITE_TAC [];
```

This may prove the goal or leave any number of subgoals. It never fails, but since it does general rewriting, it may not terminate. It has always terminated in practice. We give an example in Sect. 3.4.

Even when this doesn't prove the goal, it usually clarifies the goal by reducing it to a number of simpler subgoals. When there is a problem with the proof, for instance, missing an assumption, this disentangles what needs to be proved. If some tactic would be helpful, say expanding a definition, this can be a diagnostic aid by showing what needs to be proved and the conditions or assumptions. One can "back up" or undo the invocation, apply the necessary tactic, and proceed.

3.2 Move Quantifiers Outward

Complex inference rules may leave deeply buried quantifiers in the goal. It can be hard to even determine the scope of quantification. LIFT_QUANT_TAC combines all available conversions to move universal and existential quantifiers as far outward as possible. There universal quantifiers can be stripped and existential quantifiers can have witnesses provided in one step, rather than encountering them at different, odd times in the proof. Here is the definition.

```
val LIFT_QUANT_TAC =
 CONV_TAC (REDEPTH_CONV (
    AND_EXISTS_CONV ORELSEC AND_FORALL_CONV ORELSEC
    OR_EXISTS_CONV ORELSEC OR_FORALL_CONV ORELSEC
    LEFT_AND_EXISTS_CONV ORELSEC LEFT_AND_FORALL_CONV ORELSEC
    LEFT_IMP_EXISTS_CONV ORELSEC LEFT_IMP_FORALL_CONV ORELSEC
    LEFT_OR_EXISTS_CONV ORELSEC LEFT_OR_FORALL_CONV ORELSEC
    RIGHT_AND_EXISTS_CONV ORELSEC RIGHT_AND_FORALL_CONV ORELSEC
    RIGHT_IMP_EXISTS_CONV ORELSEC RIGHT_IMP_FORALL_CONV ORELSEC
    RIGHT_OR_EXISTS_CONV ORELSEC RIGHT_OR_FORALL_CONV));
```

This tactic never fails. We have used it after undischarging all assumptions to simplify all of them at once. See Sections 3.3 for an example.

3.3 Undischarge a Selected Assumption

Handling assumptions can be difficult, especially if one is trying to write a reusable proof. Proofs are less sensitive to change if assumptions are selected by a predicate or filter rather than by exact match or position. (A program to generate filters from an assumption list is given in [2].) FILTER_UNDISCH_TAC undischarges an assumption which matches an arbitrary predicate.

```
val FILTER_UNDISCH_TAC fp =
    let fun hfp t = fp (concl t) handle _ => false
    in
        ASSUM_LIST (fn th1 =>
               UNDISCH_TAC ((concl o hd) (filter hfp th1)))
    end;
```

This raises an exception if no assumption matches. The user supplies a term predicate to the tactic. For instance, the following undischarges the first assumption where the right hand side (#rhs) of an equality (dest_eq) is 0, that is, $\ldots = 0$. (Term delimiters are --‘ and ‘-- in HOL.)

```
  e (FILTER_UNDISCH_TAC (fn t => (#rhs o dest_eq) t = (--‘0‘--)));
```

The following example comes from a proof of information integrity. The filter function looks for an assumption which is universally quantified.

```
       preFSS inode (getFile SYS_FileSystem′ inode)
       ______________________________
       ∀inode.
           (inode = inodeOf (deref ^FP))∨
                preFSS inode (getFile SYS_FileSystem′ inode)
       ¬(inode = inodeOf (deref ^FP))

   − e (FILTER_UNDISCH_TAC (fn t => is_forall t));

   1 subgoal:
       (∀inode.
           (inode = inodeOf (deref ^FP))∨
                preFSS inode (getFile SYS_FileSystem′ inode)) ⇒
       preFSS inode (getFile SYS_FileSystem′ inode)
       ______________________________
       ¬(inode = inodeOf (deref ^FP))
```

The following tactic proves the goal.

```
e (LIFT_QUANT_TAC THEN EXISTS_TAC (--‘inode:num‘--) THEN
        ASM_REWRITE_TAC []);
```

3.4 Undischarge All Assumptions

The tactic UNDISCH_ALL_TAC undischarges all assumptions. It is helpful when one needs to manipulate all the assumptions at once. Here is the definition.

```
val UNDISCH_ALL_TAC =
    REPEAT (FIRST_ASSUM (fn thm => UNDISCH_TAC (concl thm)));
```

This tactic leaves the original goal, but with all assumptions undischarged. The following extended example comes from the proof of confidentiality of a call to `fprintf()`. We are proving that the precondition (from the previous statement's postcondition) implies the required precondition for `fprintf()`. This example also shows the use of `LIFT_QUANT_TAC` and `STRIP_THEN_REWRITE_TAC`.

```
nonConfidential (getFile SYS_FileSystem′ SYS_stdout)
————————————————————————
FP = ^FP
¬(inodeOf (deref FP) = SYS_stdout)
∀inode.(inode = SYS_stdout) ⇒
        nonConfidential (getFile SYS_FileSystem inode)
C_Result16 > 0
∀inode.(¬(inode = inodeOf (deref FP))∨
    (∃prev.((inode = SYS_stdout) ⇒ nonConfidential prev)∧
        (appendFile (printfSpec "%s %s %s %s %d " vargs)
        prev = getFile SYS_FileSystem′ inode)))∧
    ((inode = inodeOf (deref FP))∨
    ((inode = SYS_stdout) ⇒
        nonConfidential (getFile SYS_FileSystem′ inode)))
inode = SYS_stdout

- e (UNDISCH_ALL_TAC);

1 subgoal:
(FP = ^FP) ⇒ ¬(inodeOf (deref FP) = SYS_stdout) ⇒
(∀inode.(inode = SYS_stdout) ⇒
    nonConfidential (getFile SYS_FileSystem inode)) ⇒
C_Result9 > 0 ⇒
(∀inode.(¬(inode = inodeOf (deref FP))∨
    (∃prev.((inode = SYS_stdout) ⇒ nonConfidential prev)∧
        (appendFile (printfSpec "%s %s %s %s %d " vargs)
        prev = getFile SYS_FileSystem′ inode)))∧
    ((inode = inodeOf (deref FP))∨
    ((inode = SYS_stdout) ⇒
        nonConfidential (getFile SYS_FileSystem′ inode)))) ⇒
(inode = SYS_stdout) ⇒
nonConfidential (getFile SYS_FileSystem′ SYS_stdout)

- e (LIFT_QUANT_TAC);

1 subgoal:
∃inode′ inode″.
∀prev.(FP = ^FP) ⇒ ¬(inodeOf (deref FP) = SYS_stdout) ⇒
    ((inode′ = SYS_stdout) ⇒
    nonConfidential (getFile SYS_FileSystem inode′)) ⇒
```

$$
\begin{array}{l}
\texttt{C\_Result9} > 0 \Rightarrow \\
(\neg(\texttt{inode}'' = \texttt{inodeOf (deref FP)})\vee \\
((\texttt{inode}'' = \texttt{SYS\_stdout}) \Rightarrow \texttt{nonConfidential prev})\wedge \\
(\texttt{appendFile (printfSpec "\%s \%s \%s \%s \%d " vargs)} \\
\qquad\qquad \texttt{prev} = \texttt{getFile SYS\_FileSystem}' \texttt{ inode}''))\wedge \\
((\texttt{inode}'' = \texttt{inodeOf (deref FP)})\vee \\
((\texttt{inode}'' = \texttt{SYS\_stdout}) \Rightarrow \\
\texttt{nonConfidential (getFile SYS\_FileSystem}' \texttt{ inode}''))) \Rightarrow \\
(\texttt{inode} = \texttt{SYS\_stdout}) \Rightarrow \\
\texttt{nonConfidential (getFile SYS\_FileSystem}' \texttt{ SYS\_stdout)}
\end{array}
$$

3.5 An Arithmetic Tactic

While trying to prove theorems about array accesses, we came across goals which appeared easy, but took quite a bit of work. HOL has a number of good, basic tactics for arithmetic, but we could not find any general tactics.

DEPTH_ARITH_TAC simplifies as many arithmetic expressions as possible. This may prove the goal, but even if it doesn't, it eliminates some subexpressions so the user can concentrate on the parts which are not proved automatically. The implementation is as follows.

```
val DEPTH_ARITH_TAC = REDUCE_TAC THEN
    CONV_TAC (ONCE_DEPTH_CONV
        (ARITH_CONV ORELSEC NEGATE_CONV ARITH_CONV)) THEN
    ONCE_REWRITE_TAC [];
```

The following example is ripped from a proof that a piece of code finds the maximum in an array. (Some conjuncts were removed to make the example more readable.) Since n is modeled as a natural number, DEPTH_ARITH_TAC eliminates the $0 \leq n$ clauses.

$$
\begin{array}{l}
(\texttt{j} \leq \texttt{arSz}\wedge \\
\quad (\forall \texttt{n}.0 \leq \texttt{n} \wedge \texttt{n} < \texttt{j} \Rightarrow \\
\qquad \texttt{max} \geq \texttt{CA\_IDX(CA(CA\_FN ar)arSz)n}))\wedge \\
\texttt{j} \geq \texttt{arSz} \Rightarrow \\
(\forall \texttt{n}.0 \leq \texttt{n} \wedge \texttt{n} < \texttt{CA\_SZ ar} \Rightarrow \texttt{max} \geq \texttt{CA\_IDX ar n})
\end{array}
$$

− e (DEPTH_ARITH_TAC);

1 subgoal:

$$
\begin{array}{l}
(\texttt{j} \leq \texttt{arSz}\wedge \\
\quad (\forall \texttt{n}.\texttt{n} < \texttt{j} \Rightarrow \texttt{max} \geq \texttt{CA\_IDX(CA(CA\_FN ar)arSz)n}))\wedge \\
\texttt{j} \geq \texttt{arSz} \Rightarrow \\
(\forall \texttt{n}.\texttt{n} < \texttt{CA\_SZ ar} \Rightarrow \texttt{max} \geq \texttt{CA\_IDX ar n})
\end{array}
$$

DEPTH_ARITH_TAC is somewhat inefficient since for every expression, it tries to prove that the expression is true with ARITH_CONV, then if that fails, that it is false with NEGATE_CONV ARITH_CONV.

4 Tactics to Prove Goals Using Assumptions

Many times goals can be proved by instantiating assumptions or finding contradictions among assumptions. Since it may be difficult to work with assumptions in HOL [2], we find it helpful for a program to try combinations of assumptions. The three tactics in this section provide a general way to automatically manipulate assumptions to prove a goal. The first two tactics, `ESTAB_TAC` and `INCONSIST_TAC` are somewhat specific but share the same mechanism. The last one, `SOLVE_TAC`, builds on the first two for a more general tactic.

4.1 Establish a Term From the Assumptions

`ESTAB_TAC` adds the term operand as an assumption if it is provable from the assumptions. The core is a utility, `establish`, which returns a theorem of the form $a_1 \wedge \ldots \wedge a_n \Rightarrow tm$. If `establish` can prove an appropriate theorem, `ESTAB_TAC` uses it to add the term. This tactic fails if the term cannot be added.

Given a term and list of assumptions, `establish` tries different combinations of assumptions to prove the term. It tries `ARITH_CONV` and `TAUT_CONV` to prove the resulting theorem. For efficiency, `establish` tries to prove the term from each assumption, then pairs of assumptions, then triples. It only tries pairs and triples of assumptions if they share free variables.

Suppose you have the following goal.

$$\begin{array}{l} \mathtt{j} > 0 \\ \hline \mathtt{j} \geq \mathtt{arSz} \\ \forall \mathtt{n}.\mathtt{n} < \mathtt{j} \Rightarrow \mathtt{max} \geq \mathtt{f\ n} \\ \mathtt{max} = 0 \\ \mathtt{arSz} > 0 \\ \mathtt{P\ ar} = \mathtt{arSz} \end{array}$$

Some inspection shows that we could establish $\mathtt{j} > 0$ from $\mathtt{j} \geq \mathtt{arSz}$ and $\mathtt{arSz} > 0$. The following tactic proves the above goal.

```
e (ESTAB_TAC j > 0);
```

Without `ESTAB_TAC` the proving tactic is considerable longer, more sensitive to changes, and less clear.

```
e(IMP_RES_TAC (prove(j ≥ arSz ∧ arSz > 0 ∧ j ≤ arSz ⇒ j > 0, ARITH_TAC)));
```

4.2 Find an Inconsistency in the Assumptions

`INCONSIST_TAC` tries to prove a goal by finding an inconsistency in the assumptions. It fails if it cannot prove the goal.

The implementation is to try to establish `F` (false) (see `ESTAB_TAC` for details), then prove the goal since false implies anything (using `CONTR_TAC`). If it cannot establish F, it adds assumptions which follow from equalities (e.g., $a = b$) and other assumptions until it finds a matching inequality (e.g., $\sim(a = b)$), which is an inconsistency. The idea and implementation of this second approach is due to Robert Beers (beers@lal.cs.byu.edu).

Consider the following goal. Inspection suggests a proof by contradiction using the assumptions `n < j` and `j = 0` since `n` and `j` are natural numbers. `INCONSIST_TAC` proves this goal.

```
someFunction n = 0
____________________________
n < j
0 ≤ n
j = 0
```

4.3 Prove Several General Ways

The tactic `SOLVE_TAC` heavily uses `ESTAB_TAC` and `INCONSIST_TAC` to solve a goal. It tries a series of approaches and specialized tactics to prove the goal. We chose the approaches from situations which arose in proving software properties. The different ways are:

1. Establish the goal from the assumptions.
2. Prove an inconsistency in the assumptions.
3. If the goal is `a = b`, establish equalities to unify `a` and `b`.
4. If an assumption is `a ⇒ b`, establish `a` and the unifiers for `b` and the goal.

It fails if it cannot prove the goal.

`SOLVE_TAC` proves the following goals automatically.

```
P d
____________________________
a
a ⇒ P c
c = d


nonConfidential (getFile SYS_FileSystem SYS_stdout)
____________________________
inode = SYS_stdout
∀inode.(inode = SYS_stdout) ⇒
        nonConfidential (getFile SYS_FileSystem inode)
```

Because of their modular construction, these tactics can easily be improved. For instance, a version of `SOLVE_TAC` could take user's conversions, in the spirit of variations on `REWRITE_TAC`. Also `SOLVE_TAC` can be extended to handle a broader class of goals and assumptions and could memoize its attempts to unify. The `establish` routine could be more selective about which groups of assumptions to try and also do more preprocessing.

5 Experience

Our verification of a secure web server [1], consists of a total of about 720 tactic invocations in about 2,600 lines of tactics and comments. (`tac1 THEN tac2` counts as two invocations.) About 17% of the invocations are `STRIP_THEN_REWRITE_TAC`, and 6% are our other new tactics; details are in Table 1.

Tactic	Number of uses
STRIP_THEN_REWRITE_TAC	120
SOLVE_TAC	29
FILTER_UNDISCH_TAC	10
UNDISCH_ALL_TAC	5
LIFT_QUANT_TAC	2

Table 1. Uses of tactics

Although efficiency was not the goal, these tools are quick. On an HP 9000 the largest goals in our verification, which are over two hundred printed lines, took under two seconds for `WHY_NOT()` to analyze. `DEPTH_ARITH_TAC` took less than eight seconds on the largest goals. The example in Sect. 3.4 with invocations of `SOLVE_TAC` takes about 2.5 seconds.

Sources are available at the following URL. `ESTAB_TAC` and `INCONSIST_TAC` are in `establish.sml`, and `SOLVE_ TAC` is in `solveTac.sml`. The file `whynot.sml` contains `WHY_NOT()`. The rest of the tactics are in `utilities.sml`.

http://hissa.ncsl.nist.gov/~black/Source/

6 Conclusions

We have presented several generally applicable theorem proving tools. The tools include a program to analyze goals for problems which may not be caught because of HOL's type inference, tactics to simplify goals, and tactics to automatically pick out assumptions to advance a proof.

Arguably none of these tools is a breakthrough, but the group of them automates many tedious parts of proofs. They also provide a framework for incremental work to make incremental improvements in more automated proofs. These tools help to make theorem proving a little easier for new or casual users, reduce the amount of learning needed to get results, and handle details so the user can do proofs at a slightly higher level.

Acknowledgment

We thank Robert Beers for his inspiration.

References

1. Paul E. Black. *Axiomatic Semantics Verification of a Secure Web Server*. PhD thesis, Brigham Young University, Provo, Utah, USA, February 1998.
2. Paul E. Black and Phillip J. Windley. Automatically synthesized term denotation predicates: A proof aid. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications (HOL '95)*, volume 971 of *Lecture Notes in Computer Science*, pages 46–57. Springer-Verlag, 1995.
3. Graham Collins. A proof tool for reasoning about functional programs. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs '96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 109–124. Springer-Verlag, 1996.
4. John Harrison. Optimizing proof search in model elimination. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 313–327, New Brunswick, NJ, 1996. Springer-Verlag.
5. Zohar Manna and Richard Waldinger. The logic of computer programming. *IEEE Transactions on Software Engineering*, SE-4(3):199–229, May 1978.
6. Ernest Nagel and James R. Newman. *Gödel's Proof.* New York University Press, 1958.
7. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepkar Kapur, editor, *11th Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, June 1992.
8. Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

A HOL Syntax and Conventions

The following shows how a goal is displayed.

```
∃d.Pd
____________________________
a ∧ klear′ ∨ ^qq
∀inode.(inode = SYS_stdout) ⇒
        nonConfidential (getFile SYS_FileSystem inode)
```

The goal is printed above the underscore line, and all hypotheses are printed below it. Types are by default not displayed, however the user can turn on type printing.

Unbound variables are assumed to be universally quantified. Names may include underscores (_) and primes ('). Antiquotation or program (SML) variable interpolation is introduced by a caret (^).

Function application is implied (no parentheses are needed), and functions may be curried. For instance, `getFile SYS_FileSystem inode` means the function `getFile` applied to arguments `SYS_FileSystem` and `inode`.

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