Toward a Verified Generic Prooftool

Pieter Audenaert

University of Gent, Belgium

Abstract. In this short paper we present an informal report on the effort of the development, specification and verification of a generic Higher-Order-Logic prooftool. Some problems that had to be coped with are discussed in detail.

1 TAMTAM: a Generic Prooftool

TAMTAM ("TAbleau-Methods To Analyze Models") is a tableau-style prooftool. All logic-rules that are to be used are described in a logic-file, defining that particular logic. One has to provide the syntax-rules for the formulas and the tableau-style rules that are applicable on these formulas. As such, there is a complete freedom to define new logics. Since TAMTAM is programmed in the ISO-standard for Prolog [3], one has to define logics based on Prolog-style predicates, thus a basic knowledge of Prolog is absolutely necessary.

Our aim is a generic tool that is specified and verified in a formal way. A lot of prooftools are available, and most of them are for free, such as PVS, HOL, linTAP, 3TAP, PTTP or leanTAP. But when the aim is to verify the prooftool, one has to specify, verify and implement the prooftool at the same time.

TAMTAM consists of three main parts: the core (only 5% of the total code), a standard library (65%) and the interface (30%). Of course, the core is the most important part. We specified and verified this core making use of the package PVS [5]). Thus we had to write a PVS-specification for the corresponding Prolog-code. Prolog programs have both a declarative and a procedural meaning. The declarative (or descriptive) meaning is concerned with the objects and their relationships as defined by the program. The procedural meaning is concerned with how and in what order such relationships are evaluated to obtain a solution.

Ideally, programmers should restrict their attention to the declarative meaning and leave the matters of the efficiency of the search processes to the Prolog-implementation. That leads to much shorter and more easily understood programs. Unfortunately, in order to specify and verify the Prolog-code in PVS, one has to take care of the order of the evaluation and so the purely declarative approach is not sufficient. We will discuss the most important difficulties we encountered.

2 Verifying TAMTAM in PVS

As software packages become larger, it is more difficult to understand them completely. Small bugs can remain hidden for years, and at a critical point cause a lot of harm. Therefore, programs should be specified in some way, and formally verified to know for sure that they will work the way we expect.

PVS [5] has become a commonly used tool for specifying and verifying soft- and hardware. It runs in unix-environments, and it is for free. The source is not open and to program add-ons, one should know some LISP. We used PVS (v2.2) to prove that TAMTAM works correctly. As already said, PVS is mainly operational inspired, and we had some problems to specify a Prolog-environment.

The development of TAMTAM was done in two stages: first, we made a prototype and tried to specify and verify it. At this stage, we learned a lot about how to implement algorithms in Prolog in a way that they are easily translated to PVS. Then, these techniques were used in the final design, and at this stage we could benefit from the results of the early tests. We now discuss the two stages of the work:

2.1 Stage 1: a PVS-Specification for Prolog

When specifying Prolog-programs, one has to start with a specification of Prolog itself. In its essence, Prolog consists of the following parts: the syntax of terms, substitution and unification, and the execution model (resolution and backtracking).

Actually, giving a PVS-specification for all these parts is tantamount to specifying a Prolog-compiler as described in the Prolog reference-manual [3]. For example, verified Prolog-compilers can be found in [4] and [6]. This is certainly unnecessary and therefore, we developed a simplified Prolog-version, customized to PVS.

Prolog-Terms in PVS. First of all, one has to pay attention to the typing of Prolog-terms in PVS. A BNF-definition of (a part of) the types in ISO-Prolog is given as:

```
<Term>          ::= <Var>|<Atom>|<Integer>|<Float>|<Compound_Term>
<Compound_Term> ::= <Atom>(<Term>{,Term}*)
```

Recursive functions like this are difficult to deal with in PVS, and some Type-Check-Conditions (TCC's or Proof-Obligations) are generated, but some work solved this problem. Also, note that (in ISO-Prolog) a single variable is considered to be a term. Instead, we will use a different notion of variables, by lifting them up to PVS-variables. This is illustrated by the following example.

> Consider a clause `foo(X).` in a program. This means that for all `X` the predicate `foo` holds, so that we can substitute any term `t` for `X` to get a true clause `foo(t)`. This implies that we have to specify in PVS the operation of substituting terms, unifying terms, the most general unifier of two terms, etc...
>
> This involves a lot of work which can be circumvented by removing the variables from the BNF-definition, and using PVS-variables, ranging over the Prolog-terms instead. So we write down a fact as `FORALL (X: pterm): foo(X)=ptrue`, where `ptrue` stands for "true" at the Prolog-level. Note that this is equivalent to ISO-Prolog, but poses some problems when we have to prove partially instantiated goals. This is solved by using the corresponding universal quantifiers in the PVS-theorems and axioms.

This way, we developed a specification for the Prolog-syntax equivalent to that in the reference-manual [3]. In the same way, we solved the other problems concerning substitution and unification.

The Prolog Execution Model. Now one has to deal with the Prolog execution model consisting of resolution and backtracking. We learned from the work at this stage that specifying code that is relying heavily on backtracking is very difficult. Therefore, we redesigned the core (see stage 2) such that it can be described completely in terms of pre- and postconditions. This way, our Prolog-code is its own specification, and we can achieve a higher level of abstraction. The techniques used to accomplish this are described further. At this stage we can already say that in order to guarantee that the declarative and procedural meanings of the program are the same we consider a cut free implementation of the core.

In fact, the relevant part is semi-deterministic, and we programmed a special-purpose tool (called "pl2pvs") to translate parts of the core automatically to PVS-specification code.

Some Generated Code. We will now discuss an example of generated code, based on the use of "pl2pvs". First, we give a part of the type-definitions of Prolog. We start with a Prolog-term. In PVS, this is specified as an abstract datatype `pterm`. Remember that we use no variables at the Prolog-level.

```
pterm: DATATYPE
BEGIN
  ptatom(a: patom): ptatom?
  ptinteger(i: pinteger): ptinteger?
  ptfloat(f: pfloat): ptfloat?
  ptcompound(at: patom, l: (cons?[pterm])): ptcompound?
END pterm
```

Here `ptatom`, `ptinteger`, `ptfloat` and `ptcompound` are constructors that are used to generate "things" of the type `pterm`. The predicates called `ptatom?`, `ptinteger?`, `ptfloat?` and `ptcompound?` are recognizers, used to determine which sort of `pterm` we are dealing with. Note that the constructor of a compound term `at(a_1,a_2,...,a_n)` needs a Prolog-atom called "at", and a list of `pterm`s called "l", as expressed by `(cons?[pterm])` (this means "the type of lists of Prolog-terms"). Some other definitions and axioms are needed to specify the Prolog-types, but we don't want to go into details here.

Next, we define a Prolog-predication (a "callable" term):

```
ppredication: DATATYPE
BEGIN
  ppatom(a: patom): ppatom?
  ppcompound(at: patom, l: (cons?[pterm])): ppcompound?
END ppredication
```

After specifying a PVS-function `peval` (which evaluates predications to `ptrue` or `pfail`), we define the program itself as a collection of PVS-axioms, like this (note that `(:` and `:)` are used to create a list containing one element `X`):

```
prog: AXIOM
  FORALL (X: pterm):
      (peval(ppcompound(multiple_of_2,(:X:)))=ptrue
    AND
       peval(ppcompound(multiple_of_3,(:X:)))=ptrue)
    IMPLIES peval(ppcompound(multiple_of_6,(:X:)))=ptrue
```

This, of course, stands for the Prolog-clause

```
multiple_of_6(X):-
  multiple_of_2(X),
  multiple_of_3(X).
```

2.2 Stage 2: Final Design

After developing this framework for specifying Prolog-programs in PVS, we now focus on a few of the critical decisions we had to make, and some of the techniques used in the final design, to make verification feasible.

Object-oriented programming: Object-oriented software is all about "objects". An object is a "black box" which receives and sends messages, and contains "code" (sequences of computer instructions) and "data" (information which the instructions operate on). An object is defined via its "class", which determines all the properties of an object. Objects are individual "instances" of a class. Hence proving facts about objects and their methods (i.e. the action that a message carries out) is a common activity in OO software design. These techniques are heavily used in TAMTAM. We will now develop Prolog-code for such an abstract object `node` which is easy to translate to PVS-code, while explaining other techniques.

Chaining Predicates: People starting to learn Prolog often find it very confusing that one cannot assign a value to a variable. Indeed, backtracking can always undo unification. First of all, we therefore designed the Prolog-predicates as follows: `foo(+Input_1,...,+Input_n,-Output_1,...,-Output_m)` (`+` and `-` are used by convention to distinguish between input and output). When chaining predicates like this, one has the effect of passing values from one predicate to another, provided they don't fail. This is easily translated to PVS.

Semi-Deterministic Predicates: Secondly, the predicates are designed in a semi-deterministic way, which means that, although the function can possibly backtrack, theorems can be proven about them, in terms of pre- and postconditions. This way we can handle the problem of backtracking predicates in PVS.

Making use of these properties, one can use techniques from object-oriented-programming to introduce objects and their methods. There are only three types of objects in TAMTAM: proofs, proof-trees, and proof-tree-nodes. Their respective hierarchy is assumed to be clear. In TAMTAM, `Node`s are defined as an abstract datatype containing information on nodes in a tableau. For example, it contains a list of antecedent-formulas, and a number describing its position in the tree:

```
node_begin_definition.  % Begin the definition of an object 'node'
                        % and it's methods.

node_definition(X):-    % The "functor"-goal is a constructor for
  functor(X,node,8).    % nodes, this creates a term "X"
                        % containing 8 pieces of information.

node_aformulas(X,Y):-   % A selector for the list of antecedent-
  node_definition(X),   % formulas. The "arg"-goal selects the
  arg(2,X,Y).           % second argument of "X", and instantiates
                        % the variable "Y" to it.

node_number(X,Y):-      % A selector for the number of the node.
  node_definition(X),
  arg(6,X,Y).

...                     % Some other selectors and methods.
```

Now we will develop a copy-constructor for an object `node`. Most of the time we will use this to copy information from one variable `Input` to another called `Output`, but in the mean time changing the value in `X` to another value `Val`:

```
node_copy(Input,X,Val,Output):-
  node_copy_except(X,Input,Output), % Copy everything from Input
                                    % to Output, except for 'X'.
  atom_concat('node_',X,Selector),  % Choose the correct selector,
  Put=..[Selector,Output,Val],      % and construct the goal,
  Put.                              % which is called.

node_copy_except(X,Input,Output):-  % Copy everything except 'X'
  node_definition(Output),          % making use of selectors.
  node_copy_except_routine(X,Input,Output,aformulas),
  node_copy_except_routine(X,Input,Output,number),
  ...                               % A lot of other things are
                                    % to be copied too...

node_copy_except_routine(X,_,_,Y):- % Don't copy the chosen value.
  X=Y,!.
node_copy_except_routine(_,Input,Output,Y):- % Now 'X' is
  atom_concat('node_',Y,Selector), % different from 'Y', and we
  Get=..[Selector,Input,Arg],      % construct a goal to select
  Get,                             % the value. This goal is
  Put=..[Selector,Output,Arg],     % called and the value is put
  Put.                             % in the right place.
```

After developing the class of `Nodes`, we can immediately start proving properties about it. For example, suppose we copy `Input` to `Output` except for the `number` which is set to the Prolog-list `[1,2,3]`, then we can prove in PVS that applying `node_copy(Input,number,[1,2,3],Output)` implies that the `formulas` remain unchanged, but the `number` of `Output` is now `[1,2,3]` (note that the list is represented in PVS by `(:1,2,3:)`):

```
node_th1: THEOREM
  FORALL (node1,node2: NODES):
      node_copy(node1,number,(:1,2,3:),node2)
    IMPLIES
      node_formulas(node1)=node_formulas(node2)

node_th2: THEOREM
  FORALL (node1,node2: NODES):
      node_copy(node1,number,(:1,2,3:),node2)
    IMPLIES
      node_number(node2)=(:1,2,3:)
```

Abstraction: As already said, abstraction leads to positive results easier. As an example, we will discuss the code to open and close streams in TAMTAM, and show how good Prolog-programming can result in better PVS-code. Consider the ISO-predicate to open streams: `open(+File,+Mode,-Stream,+Options)`. Now suppose you want to open a file, do something with it, and then close it:

```
goal:-
  open('foo','read',Stream,[]),   % Open a stream for this file.
  do_something(Stream),           % Do something with it.
  close(Stream).                  % Close the stream.
```

This seems fine, but what happens when the predicate `do_something/1` suddenly fails completely unexpected (due to a disk error for example)? As a result, the main `goal` will fail too, and we will lose our reference `Stream` to our file, which remains open. Thus we should abstract the concept of streams in our PVS-specification.

Streams are specified as a datatype, from which one can extract data, and nothing is said about opening or closing them. Of course, the Prolog-code accomplishing this is verified separately. Frequently used, small and verified code-snippets are collected in a sort of standard library. This way, we avoid being distracted in the main PVS-proof by details.

Now suppose again that we want to invoke the goal `do_something/1` on the file `foo`. Then we call `checkstream('foo','read',[],do_something(_))`. This will open a read-stream for the file `foo`, and call `do_something(Stream)` with `Stream` properly instantiated. When finished, `checkstream` will close the stream and it will succeed if `do_something` succeeds, and fail when `do_something` fails. This is the code:

```
checkstream(File,Mode,Options,Goal):-
  Goal=..Goal_List,                % Unify last argument of Goal
  last(Stream,Goal_List),          % with the Stream.
  open(File,Mode,Stream,Options),  % Open it.
  checkstream_go(Stream,Goal),!.   % Call subroutine.

checkstream_go(Stream,Goal):-
  Goal,                            % Call Goal, and if it succeeds
  close(Stream,[force(true)]).     % close the Stream.
checkstream_go(Stream,_):-         % Our Goal failed
  close(Stream,[force(true)]),     % but nevertheless close stream.
  fail.                            % Then fail.
```

Other techniques are used to make our PVS-specification more abstract, but we cannot explain everything here.

3 Conclusion

As a conclusion, we can say that is was very difficult to develop the verified prooftool TAMTAM, because we had to think continuously about how to code Prolog-algorithms in a way that we could translate it in PVS-code. A lot of techniques are employed to achieve this goal. Once this was done, we could immediately start proving interesting properties of the tool. The core is not verified completely yet, but this will be done soon.

References

1. Audenaert P., paudenae@cage.rug.ac.be, http://cage.rug.ac.be/~paudenae
2. Bratko I., Prolog Programming for Artificial Intelligence, Addison-Wesley, 2001.
3. Deransart P., Ed-Dbali A., Cervoni L., Prolog: The Standard, Springer-Verlag, New York, 1996.
4. Hanus M., Formal Specification of a Prolog Compiler, Lecture Notes in Computer Science 348, Programming Languages Implementation and Logic Programming, Deransart P., Lorho B., Maluszynski J. (Eds.), Springer-Verlag, New York 1989.
5. PVS, http://pvs.csl.sri.com/
6. Russinoff D. M., A verified Prolog compiler for the Warren abstract machine, MCC Technical Report Number ACT-ST-292-89, Microelectronics and Computer Technology Corporation, Austin, TX, July 1989.