Proof certificates in PVS

Frédéric Gilbert

Proof certificates in PVS

Frédéric Gilbert*

École des Ponts ParisTech, Inria, CEA LIST
frederic.a.gilbert@inria.fr

Abstract. The purpose of this work is to allow the proof system PVS to export proof certificates that can be checked externally. This is done through the instrumentation of PVS to record detailed proofs step by step during the proof search process. At the current stage of this work, proofs can be built for any PVS theory. However, some reasoning steps rely on unverified assumptions. For a restricted fragment of PVS, the proofs are exported to the universal proof checker Dedukti, and the unverified assumptions are proved externally using the automated theorem prover MetiTarski.

1 Introduction

Given the complexity of proof assistants such as PVS, external verifications become necessary to reach the highest levels of trust in its results. A possible way to this end is to require the system to export certificates that can be checked using third-party tools. The purpose of this work is to instrument PVS to export certificates that can be verified externally.

This approach is comparable to the OpenTheory project [3], in which the higher order logic theorem provers HOL Light, HOL4, and ProofPower are instrumented to export verifiable certificates in a shared format. In HOL Light, HOL4, and ProofPower, the detail of each reasoning step is expressed using a small number of simple logical rules, which are used as a starting point to the generation of OpenTheory certificates. As this is not the case in PVS, the whole proof system needs to be instrumented to generate complete certificates. At the current stage of this work, this instrumentation is not complete, leading to the presence of unverified assumptions in the generated certificates. For a restricted fragment of PVS, the proof certificates are exported to the universal proof checker Dedukti [5], and the unverified assumptions are proved externally using the automated theorem prover MetiTarski [1].

In PVS [4], the proof process is decomposed into a succession of proof steps. These proof steps are recorded into a proof trace format, the `.prf` files. These proof traces can be used to rerun and verify a proof, but only internally. In order to check these proof traces externally, one would have to reimplement PVS proof mechanisms almost entirely.

* This work has been completed as part of two visits to the National Institute of Aerospace (NIA) and the NASA Langley Research Center under NASA/NIA Research Cooperative Agreement No. NNL09AA00A

The purpose of the proof certificates presented in this work is to check PVS proofs externally using small systems. To this end, we present a decomposition of PVS proof steps into a small number of atomic rules, which are easier to encode into a third-party system than the original proof steps. The proof certificates are built on these atomic rules, and can be checked without having to reimplement PVS proof steps.

These atomic rules are defined as a refinement of an intermediate decomposition of proof steps which is already present in PVS. This intermediate decomposition is based on a specific subset of proof steps, the primitive rules. In PVS, every proof step, including defined rules and strategies, can be decomposed as a sequence of primitive rules. As any primitive step is a proof step, this intermediate level of decomposition can be formalized in the original format of `.prf` proof traces. In fact, such a decomposition can be performed using the PVS package Manip [2], in which the instruction `expand-strategy-steps` allows one to decompose every proof step into a succession of primitive rules.

However, this intermediate decomposition is not sufficient to make proof traces verifiable externally using small systems. Indeed, the complexity of PVS proof mechanisms lies for the largest part in the primitive rules themselves. In particular, the implementation of primitive rules is one order of magnitude larger than the implementation of strategies. For instance, the primitive rule `simplify` hides advanced reasoning techniques, including simplifications, rewritings, and Shostak's decision procedures.

In order to provide a refinement of the primitive rule decomposition, we modify PVS directly to record reasoning at a higher level of precision. The main part of this modification is done in the code of the primitive rules themselves. This instrumentation doesn't affect the reasoning in any way besides some slowdown due to the recording of proofs. In particular, it doesn't affect the emission of `.prf` proof traces, which continue to be used internally to rerun proofs as in the original system.

The coherence of a PVS theory is based on both reasoning and typing. At the current stage of this work, the proof certificates are limited to reasoning. Moreover, primitive rules are not entirely instrumented, and the corresponding gaps in reasoning are completed with unverified assumptions.

In the next section, we present the formalization of proof certificates in PVS. Then, we present a first attempt to export these proofs to the universal proof checker Dedukti [5], and to export their unverified assumptions to the theorem prover MetiTarski [1].

2 Proofs certificates in PVS

2.1 Expressions and conversion

Proof are added as a new layer of abstract syntax, on top of the existing layers of PVS expressions and PVS sequents. For readability, we will denote PVS expressions as they are printed in PVS. We stress the fact that this denotation

is not faithful, as several components of PVS expressions, such as types and resolutions, are erased through PVS printing.

As several other proof systems, Dedukti is equipped with a notion of conversion, which includes, among others, β-conversion and constant definitions, which will be referred to as δ-conversion. As a consequence, it is not necessary to record the expansion of a definition or the reduction of a β-redex in Dedukti, which allows us to keep proofs compact.

Following this idea, we equip PVS expressions with a conversion, denoted $\equiv$. This conversion includes β-conversion, and non-recursive definitions, expressed as δ-rules. However, δ rules are not used for recursive definitions as this would lead to infinite reductions: instead, the expansions or contractions of recursive definitions are kept as explicit reasoning steps.

2.2 Reasoning

In PVS, internally, the formulas appearing on both sides of a sequents are recorded in a single list, where all formulas belonging to the left hand side appear under a negation. For instance, a sequent appearing as `NOT A, B` $\vdash$ `C` is recorded internally as the list `NOT NOT A, NOT B, C`. Denoting Γ the union of this list together with the list of hidden formulas, the corresponding sequent will be denoted $\vdash \Gamma$.

We equip sequents with the identification modulo permutation. In this setting, sequents correspond to multisets, and we don't need to record any exchange rule, which makes proofs more compact. On top of this layer of sequents, we use the following rules, which are presented modulo conversion $\equiv$.

Structural rules

$$\frac{}{\vdash \Gamma, \texttt{A, NOT A}} \qquad \frac{\vdash \Gamma, \texttt{A} \qquad \vdash \Gamma, \texttt{NOT A}}{\vdash \Gamma} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \texttt{A}} \qquad \frac{\vdash \Gamma, \texttt{A, A}}{\vdash \Gamma, \texttt{A}}$$

Propositional rules

$$\frac{}{\vdash \Gamma, \texttt{TRUE}} \qquad \frac{\vdash \Gamma, \texttt{NOT TRUE}}{\vdash \Gamma} \qquad \frac{\vdash \Gamma, \texttt{FALSE}}{\vdash \Gamma} \qquad \frac{}{\vdash \Gamma, \texttt{NOT FALSE}}$$

$$\frac{\vdash \Gamma, \texttt{A} \qquad \vdash \Gamma, \texttt{B}}{\vdash \Gamma, \texttt{A AND B}} \qquad \frac{\vdash \Gamma, \texttt{NOT A, NOT B}}{\vdash \Gamma, \texttt{NOT (A AND B)}}$$

$$\frac{\vdash \Gamma, \texttt{A, B}}{\vdash \Gamma, \texttt{A OR B}} \qquad \frac{\vdash \Gamma, \texttt{NOT A} \qquad \vdash \Gamma, \texttt{NOT B}}{\vdash \Gamma, \texttt{NOT (A OR B)}}$$

$$\frac{\vdash \Gamma, \texttt{NOT A, B}}{\vdash \Gamma, \texttt{A IMPLIES B}} \qquad \frac{\vdash \Gamma, \texttt{NOT B} \qquad \vdash \Gamma, \texttt{A}}{\vdash \Gamma, \texttt{NOT (A IMPLIES B)}} \qquad \frac{\vdash \Gamma, \texttt{A}}{\vdash \Gamma, \texttt{NOT NOT A}}$$

$$\frac{\vdash \Gamma, \texttt{A IMPLIES B} \qquad \vdash \Gamma, \texttt{B IMPLIES A}}{\vdash \Gamma, \texttt{A IFF B}}$$

$$\frac{\vdash \Gamma, \texttt{NOT (A IMPLIES B), NOT (B IMPLIES A)}}{\vdash \Gamma, \texttt{NOT (A IFF B)}}$$

$$\frac{\vdash \Gamma, \texttt{A IMPLIES B} \qquad \vdash \Gamma, \texttt{NOT A IMPLIES C}}{\vdash \Gamma, \texttt{IF(A, B, C)}}$$

$$\frac{\vdash \Gamma, \texttt{NOT (A AND B)} \qquad \vdash \Gamma, \texttt{NOT (NOT A AND C)}}{\vdash \Gamma, \texttt{NOT IF(A, B, C)}}$$

Quantification rules

$$\frac{\vdash \Gamma, \texttt{A}}{\vdash \Gamma, \texttt{FORALL (x : T) : A}} \qquad \frac{\vdash \Gamma, \texttt{NOT A[t/x]}}{\vdash \Gamma, \texttt{NOT FORALL (x : T) : A}}$$

$$\frac{\vdash \Gamma, \texttt{A[t/x]}}{\vdash \Gamma, \texttt{EXISTS (x : T) : A}} \qquad \frac{\vdash \Gamma, \texttt{NOT A}}{\vdash \Gamma, \texttt{NOT EXISTS (x : T) : A}}$$

Equality rules

$$\frac{}{\vdash \Gamma, \texttt{t = t}} \qquad \frac{\vdash \Gamma, \texttt{t = u} \qquad \vdash \Gamma, \texttt{u = v}}{\vdash \Gamma, \texttt{t = v}}$$

$$\frac{\vdash \Gamma, \texttt{A(t)} \qquad \vdash \Gamma, \texttt{t = u}}{\vdash \Gamma, \texttt{A(u)}} \qquad \frac{\vdash \Gamma, \texttt{u = v}}{\vdash \Gamma, \texttt{f(u) = f(v)}}$$

$$\frac{\vdash \Gamma, \texttt{NOT A, u = v}}{\vdash \Gamma, \texttt{IF(A, u, t) = IF(A, v, t)}} \qquad \frac{\vdash \Gamma, \texttt{A, u = v}}{\vdash \Gamma, \texttt{IF(A, t, u) = IF(A, t, v)}}$$

Extensionality rules

$$\frac{\vdash \Gamma, \texttt{A IFF B}}{\vdash \Gamma, \texttt{A = B}} \qquad \frac{\vdash \Gamma, \texttt{t = u}}{\vdash \Gamma, \texttt{LAMBDA (x : T) : t = LAMBDA (x : T) : u}}$$

$$\frac{\vdash \Gamma, \texttt{t = u}}{\vdash \Gamma, \texttt{FORALL (x : T) : t = FORALL (x : T) : u}}$$

$$\frac{\vdash \Gamma, \texttt{t = u}}{\vdash \Gamma, \texttt{EXISTS (x : T) : t = EXISTS (x : T) : u}}$$

Extra rules

$$\frac{\vdash \Gamma, \Delta \qquad \vdash \Gamma, \Delta_1 \qquad \cdots \qquad \vdash \Gamma, \Delta_n}{\vdash \Gamma, \Delta}\ TCC$$

$$\frac{\vdash \Gamma, \Delta_1 \qquad \cdots \qquad \vdash \Gamma, \Delta_n}{\vdash \Gamma, \Delta}\ \mathit{Assumption}$$

Only the two last rules, TCC and *Assumption*, are specific to this system. The first one is due to the appearance of type-checking conditions during the proof run, for instance after giving an instantiation for an existential proposition. As typing is not checked in such proofs, this condition is not necessary, but this rule allows us to ensure that all steps of reasoning are recorded in proofs, included the reasoning steps ensuring typing constraints.

The second one, *Assumption*, is generated from all reasoning steps in PVS which haven't been instrumented yet. In practice, the use of *Assumption* doesn't imply that the corresponding reasoning gap cannot be described using the other rules. For instance, the primitive rule *bddsimp*, which calls a function outside the PVS kernel, was not instrumented. Yet, the corresponding reasoning steps could be justified using structural and propositional rules. On the other hand, the strategy *prop*, which has the same role, doesn't generate any *Assumption* rule, as the underlying primitive rules $flatten$ and *split* are both instrumented.

2.3 Proof objects

In order to record lightweight proofs, we record only the rules used in the proofs, provided with a sufficient amount of rule parameters.

For instance, the proof

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{NOT A, NOT B, A}}}{\vdash \texttt{NOT A, NOT B, NOT NOT A}}}{\vdash \texttt{NOT (A AND B), NOT NOT A}}}{\vdash \texttt{(A AND B) IMPLIES NOT NOT A}}$$

will be recorded as

```
RImplies(A AND B, NOT NOT A,
RNotAnd(A, B,
  RNotNot(A,
    RAxiom(A))))
```

where `RImplies`, `RNotAnd`, `RNotNot`, and `RAxiom` denote the rules used in the proof, and accept as argument a list of parameters followed by a (possibly empty) list of subproofs.

3 Checking PVS proofs using Dedukti and Metitarski

This part of the work is only at the stage of a first prototype. The universal proof checker Dedukti is used to verify the proof certificates. As these certificates contain unverified assumptions, the automated theorem prover MetiTarski is used to prove them externally.

3.1 Translating proofs to Dedukti

Dedukti is a dependently typed language. However, as we only record reasoning in this work, we use a translation which doesn't make PVS types appear. We declare one universal type `type` for all PVS expressions. In order to translate applications, we use a constant `apply : type -> type -> type`. Conversely, we use a constants `lambda : (type -> type) -> type` to translate lambda expressions.

A similar technique is used to translate the other constructions appearing in the rules, such as `FORALL`.

The translation from PVS proofs to Dedukti is a translation from sequent calculus to natural deduction. The use of Dedukti being based on the Curry-Howard isomorphism, a proof of a proposition `A` is expected as a term of type `A`. The main translation function takes a proof of a sequent $\vdash \texttt{A}_1, ..., \texttt{A}_n$ and a list of proof variables $h_1, ..., h_n$ to a produce a term p which has the type `FALSE` in the context $h_1 : \texttt{NOT A}_1, ..., h_n : \texttt{NOT A}_n$. This translation is based on the declaration of the rules as constants in Dedukti.

Using this main translation function, for any proposition `A` proved in PVS, and for any proof variable h, we build a proof p of type `FALSE` in the context h : `NOT A`. Then, using a rule of negation introduction together with a rule of double negation elimination, we get a proof term of type `A` in the empty context, as expected.

3.2 Checking assumptions with MetiTarski

Every rule except `Assumption` is valid in classical higher-order logic. In order to check the assumption rules as well, we use an automated theorem prover. We chose the first-order theorem prover MetiTarski for this purpose.

Using conjunctions, disjunctions and implications, every assumption rule is translated into a single proposition, which in turn is translated to the TPTP [6] format. The main issue in this translation is the presence of higher-order expressions, such as lambda terms of if-then-else expressions for instance. In this work, these terms are translated as constant symbols: the obtained expressions are correct TPTP expressions, and their validity in first-order logic ensures the validity of the original expression in higher-order logic.

4 Results

The instrumentation of PVS to build proof certificates is not restricted to any fragment of PVS. It has been tested using the arithmetic theories (`ints`) of the NASA Library `nasalib`. The generation of all certificates for the whole (`ints`) library (32 files, 268 proofs) was performed in one hour.

The exportation to Dedukti and MetiTarski has been tested on the following example:

```
induction : THEORY
   BEGIN
   f : [nat -> nat]
   nat_sum : LEMMA
       (f(0) = 0 AND (FORALL (n:nat): f(n+1) = f(n) + n + 1))
             IMPLIES FORALL (n:nat): 2 * f(n) = n * (n + 1)
END induction
```

This theorem was proved in two steps: `flatten`, and `induct-and-simplify`. The Dedukti file generated has been successfully checked by Dedukti. It contained 19 unverified assumptions. All of them have been successfully proved using MetiTarski.

References

1. Behzad Akbarpour and Lawrence Charles Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.
2. Ben L Di Vito. Manip user's guide, version 1.3. 2011.
3. Joe Hurd. The opentheory standard theory library. *NASA Formal Methods*, pages 177–191, 2011.
4. Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.
5. Ronan Saillard. Dedukti: a universal proof checker. In *Foundation of Mathematics for Computer-Aided Formalization Workshop*, 2013.
6. Geoff Sutcliffe. The tptp problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337, 2009.