Square root and division elimination in PVS

Pierre Neron

Square root and division elimination in PVS

Pierre Neron

École polytechnique - Inria

Abstract. In this paper we present a new strategy for PVS that implements a square root and division elimination in order to use automatic arithmetic strategies that were not able to deal with these operations in the first place. This strategy relies on a PVS formalization of the square root and division elimination and deep embedding of PVS expressions inside PVS. Therefore using computational reflection and symbolic computation we are able to automatically transform expressions into division and square root free ones before using these decision procedures.

Introduction Proof verification systems such as PVS [7] embed proofs strategies that allow the user to deal with arithmetic problems automatically. However most of these techniques such as the use of SMT solvers [2,4] or quantifier elimination [3] are not able to manage all arithmetics operations, in particular division and mainly square roots. Being able to transform any goal or hypothesis containing square roots or divisions into an equivalent one that is free of them would allow the use of arithmetic decision procedures to resolve the current goal.

A program transformation that removes square roots and divisions from programs has been defined and proved correct in PVS, see [6]. We now aim at using this implementation of the transformation and the proof of the semantics equivalence between the input and the output formulas to define a PVS strategy [1]. This strategy, `elim-sqrt`, transforms any goal or hypothesis by eliminating square roots and divisions from it *e.g.,*

```
    {-1} x <= 1                              {-1} x <= 1
      |----                  ⟶                 |----
    {1} x <= sqrt(x)       elim-sqrt         {1} x * x - x <= 0
```

This is realized by doing a deep embedding [8] of a fragment of PVS inside PVS in order to use computational reflection for transformation computation [5]. This is a big difference with PVS or Coq `fields` strategies, that are written in the strategy language, since the size of the proof does not depend on the input terms.

1 Deep embedding

First of all we need to sketch how this transformation is specified in PVS, the complete definition can be found in [6].

Definitions The transformation in PVS is defined on programs represented in an abstract datatype `program`. It represents variables, constants, some operators, pairs, projections, variable definitions and conditional expressions:

Definition 1.1 (program abstract datatype)

```
program : DATATYPE
        value(va : variable) : value?
        const(co : constant) : const?
        uop(uop : unop, pr : program) : uop?
        bop(bop : binop, pl : program, pr : program) : bop?
        pair( pl : program, pr : program) : pair?
        fst( pr : program) : fst?
        snd( pr : program) : snd?
        letin( x : variable, body : program, scope : program) : letin?
        ift(fm : program, prt : program, prf : program) : ift?
```

The operators in the `binop` and `unop` datatypes represent $+, -, *, /, >, \geq, =, \vee, \wedge$ and $\neg$. Functions computing the type and the semantics of a `program` in a given environment are defined in PVS. The semantics of a `program` is either a failure (*e.g.*, division by 0) or a tuple of boolean and numerical values:

Definition 1.2 (Semantics function)

```
  sem(p : program, env : eval_env) : RECURSIVE prog_val
```

where `prog_val : DATATYPE`

```
            numv(re : real): numv?
            boolv(bo : bool): boolv?
            pairv(vl : prog_value, vr : prog_value): pairv?
            failv: failv?
```

and `eval_env = [variable -> prog_val]`

Given these definitions, we can now introduce the main definition of the transformation as a PVS function `elim` defined on `program`. PVS subtyping allows us to embed the preservation of the semantics in the type of this function:

Definition 1.3 (Main transformation)

```
  elim(p : program) :
      {pp : program_N_sq | preserves_semantics_no_fail(p)(pp)}
```

where `program_N_sq` *is the subtype of* `program` *without square roots and divisions and* `preserves_semantics_no_fail(p)(pp)` *the following statement:*

```
    ∀ env, nofailv(sem(p,env)) IMPLIES sem(p,env) = sem(pp,env)
```

In order to use this transformation, we have to transpose a PVS statement into this formalism, this realizes a deep embedding of a fragment of PVS inside PVS.

Deep embedding Given a proof context in PVS, we aim at transforming a statement (either a goal or an hypothesis) into an equivalent one which is free of divisions and square roots. First of all, as we can see in definition 1.1 the formalism only represents a fragment of PVS, therefore the statement we want to transform has to match this formalism. Given such a statement, we call it `S`,

the first step of this embedding is to compute the equivalent `p : program` and the corresponding evaluation environment `env` such that:

    sem(p,env) = boolv(S)

Indeed, the `variable` of the `program` type are not PVS variables but identifiers (*e.g.,* string or natural numbers), therefore we need the environment to make the link between these identifiers and their value, *i.e.,* the value of the corresponding PVS variables. From now on, given a PVS variable `x` in a statement and the corresponding environment `env`, its identifier will be the string `"X"`. These elements, the `program` and `environment`, have to be computed as their PVS string representation:

Example 1.1 (Equivalent program in environment)

```
                    p = "bop(gt,uop(sqrt,fst(var("X"))), var("Y"))"
  |--- {1}          env = "LAMBDA (z : string) :
sqrt(x'1) > y   →        IF z = "X" THEN pairv(numv(x'1),numv(x'2))
                         ELSIF z = "Y" THEN numv(y) ELSE 0 ENDIF"
```

This string representation allows us to introduce these items in the current context with some PVS prover commands.

Equivalent `program` computation Given a PVS context and a statement `S`, by using the strategy language we can access to the corresponding lisp tree structure that represents the abstract syntax of the PVS statement. Therefore if the statement matches the embedded fragment, computing the equivalent `program` can be done by decomposing this lisp structure and building the corresponding string. As most of the cases are staightforward, we only detail a few of them:

- the variable: as mentioned earlier, the variables of the `program` type are identifiers (*e.g.,* string) and we need to have a mapping beetween every PVS variable and its corresponding string identifier.
- the projections: in PVS, tuples are represented as arrays (int → element), the corresponding lisp object is a list and we need to translate it as a binary tree, *e.g.,* list (e1 e2 e3) gives `pair(e1,pair(e2,e3))` and the projection x'3 is translated into `"snd(snd(Value"X"))"`

Corresponding evaluation environment As we can see in example 1.1, the correspondence between identifiers and variables is not straightforward either. Indeed, we need to build the value corresponding to each identifier. Given an identifier `"X"` and its associated variable `x`, if `x` has a basic type, `number` or `bool`, then the semantics of `value("X")` is `x`, but if `x` is a tuple, then we need to extract its elements and build the corresponding `prog_val`. For example if `x` is a triple of type [`bool`,`real`,`real`] then the associated value `prog_val` is `pairv(boolv(x'1),pairv(numv(x'2),numv(x'3)))`.

Given a PVS statement `E`, we are able to compute the corresponding `program` `p` and environment `env`, such that `E` can be replaced by the semantics of `p`, *i.e.,*

`bo(sem(p,env))`. This allows us to work on the `program p` in order to apply the transformation.

2   Strategy definition

In this section we present how to build the strategy that transforms a current goal or hypothesis into an equivalent square root and division free one. In the `program` expressions we will avoid writing constructors that are obvious, *e.g.*, we will write `"A"` and `plus(e1,e2)` instead of `val("A")` and `bop(plus,e1,e2)`.

Strategy principles Fig. 1 describes the main steps of the `elim-sqrt` strategy:

```
     |----                    strategy         |----
    {1} sqrt(a) > b       ------------- →      {1} -b > 0
                                               {2} a > b * b
     (1) grind |                                    | (4) grind
                ↓                                    ↓
 sem(gt(sqrt("A"),"B"),env)          sem(or(gt(0,B),gt(A,times(B,B))),env)

  (2) typepred "elim"  ↘                    ↗  (3) eval-expr

            sem(elim(gt(sqrt(A),B)),env)
```
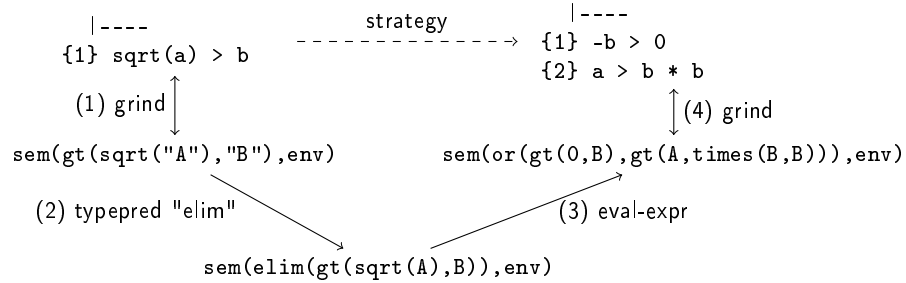
Fig. 1. `elim-sqrt` strategy outlines

(1) we introduce the equivalent `program` and environment and prove this equivalence using symbolic evaluation with `grind`
(2) using the type predicate of `elim` we apply this function to the `program`
(3) we compute the elimination using computational reflection `eval-expr`
(4) we return into the PVS language itself using symbolic evaluation of the square root and division free `program` semantics

In section 2, we gave the main steps of the transformation strategy, we will now see how these different expressions can be introduced in the PVS prover, and their equivalence proved. In this section we will assume that we have an hypothesis, `H`, we want to remove square roots from, the elimination in a positive formula (*e.g.*, a Goal) being similar.

From PVS expression to `program` datatype As mentioned in section 1 the transformation is defined using the `program` abstract datatype, the first step of the strategy is therefore to transpose the PVS statement into this datatype. In 1 we introduced a lisp function that, given a PVS statement, builds the corresponding `program`, `p` and environment `env`. The first step of the strategy is to introduce this `program` equivalent to `H` using its boolean semantics `bo(sem(p,env))`. The extraction of the boolean part of the semantics with `bo` such as the use of the type of the `elim` function will require to prove that `sem(p,env)` does not fail

and is a boolean `prog_val`, this can be done by doing a symbolic evaluation of
`sem(p,env)` but this evaluation is not very efficient. Therefore in order to do it
only once, we introduce explicitly this hypothesis with the following command:

$$\texttt{(case "boolv?(sem(p,env)) AND bo(sem(p,env))")}$$

This rule introduces a new hypothesis we first have to prove in the current con-
text. The proof of `boolv?(sem(p,env)) AND bo(sem(p,env))` only uses the
symbolic evaluation of `sem(p,env)` that produces `boolv(H)` and therefore fin-
ishes that case. Now that we have introduced `bo(sem(p,env))` equivalent to `H`,
we can delete `H` from the context.

elim function introduction We now want to eliminate square roots and di-
visions from `p`. Hence, we introduce the type of `elim(p)`, with the `typepred`
command (1), `nofail(sem(p,env))` is straightforward using the `-2` hypothesis
and thus it allows the use of the semantics equality to replace `p` by `elim(p)` (2):

```
    {-1} nofail(sem(p,env)) IMPLIES
    sem(p,env) = sem(elim(p),env)           {-1} boolv?(sem(elim(p),env))
    {-2} boolv?(sem(p,env))                 {-2} bo(sem(elim(p),env))
(1) {-3} bo(sem(p,env))               (2)   {-3} Hypothesis
    {-4} Hypothesis                             |----
       |----                                 {1} Goal
    {1} Goal
```

Computational reflection The next step is to produce the equivalent square
root and division free formula, this is done by computational reflection of `elim(p)`.
The use of this technique requires two hypotheses:

- the function, (*i.e.*, `elim`) has to be completely defined with computable struc-
  tures (*e.g.*, use list instead of sets), so there is a corresponding executable
  lisp function,
- the arguments have to be ground (do not contain any PVS variable), this is
  ensured by using identifiers to represent the original PVS variable, the link
  between these identifiers and variables being handled separately by `env`.

Therefore we can compute `elim(p)` in order to get the equivalent `program, p'`,
free of square roots and divisions with the eval-expr strategy.

Semantics evaluation From our new square root and division free `program`
`p'` we want to get the corresponding PVS expression. Therefore we have to
compute the semantics of this `program`. This is done once again by symbolic
evaluation and in the end we get a new PVS statement `H'`, equivalent to `H`, free
of square roots and divisions. Square roots and divisions being eliminated in this
hypothesis we can now continue the proof using our favorite arithmetic strategy.

Conclusion

We have described how to turn a PVS computable specification and the corresponding proof of a program transformation into a PVS strategy. We realized it by doing a deep embedding of PVS inside PVS, using symbolic evaluation to prove the correspondence between PVS and its embedding when the transformation itself uses computational reflection. This kind of embedding can be generalized for any transformation defined in PVS on an abstract datatype representing a fragment of PVS.

This strategy has been tested on various examples, from simple comparisons to more complex statements that embed variable definitions and conditional expressions. The strategy takes between 20 sec to few minutes mainly depending on the number of square roots. These results can be explained by the low performances of the PVS symbolic evaluation whereas the transformation itself that uses reflection, is almost instantaneous.

This strategy is also the first step of a larger scale transformation that aims at eliminating square roots and divisions from full PVS specifications and producing a semantics equivalence proof certificate.

References

1. M. Archer, B. D. Vito, and C. Muñoz. Developing user strategies in PVS: A tutorial. In *Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics STRATA'03*, NASA/CP-2003-212448, NASA LaRC,Hampton VA 23681-2199, USA, September 2003.
2. C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
3. G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis. *SIGSAM Bull.*, 10:10–12, February 1976.
4. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for dpll(t). In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
5. S. Lescuyer and S. Conchon. Improving coq propositional reasoning using a lazy cnf conversion scheme. In S. Ghilardi and R. Sebastiani, editors, *FroCoS*, volume 5749 of *LNCS*, pages 287–303. Springer, 2009.
6. P. Neron. A formal proof of square root and division elimination in embedded programs. In C. Hawblitzel and D. Miller, editors, *CPP*, volume 7679 of *LNCS*, pages 256–272. Springer, 2012.
7. S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In D. Kapur, editor, *CADE*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.
8. M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.